

Prozeßkoordinierung durch Verwendung von Fortsetzungsbedingungen

von
Klaus Winkelmann

Erlangen – 1984

1984 № 244

Prozeßkoordinierung durch Verwendung von Fortsetzungsbedingungen

Der Technischen Fakultät
der Universität Erlangen - Nürnberg
zur Erlangung des Grades
DOKTOR - INGENIEUR

vorgelegt von
Klaus Winkelmann

Erlangen - 1984

Als Dissertation genehmigt von der
Technischen Fakultät der
Universität Erlangen-Nürnberg

Tag der Einreichung: 9.2.1984

Tag der Promotion: 24.5.1984

Dekan: Prof. Dipl.-Ing. G. Vetter

Berichterstatter: Prof. Dr. F. Hofmann

Prof. Dr. H.J. Schneider

Meinen lieben Eltern in Dankbarkeit gewidmet.

Ich danke Herrn Professor Dr. F. Hofmann sehr herzlich für die Anregung und die engagierte Betreuung dieser Arbeit. Er hat ihr Gelingen durch verständiges Zuhören, wertvolle Hinweise und treffende Fragen erst möglich gemacht.

Für die Übernahme des Zweitgutachtens trotz Zeitnot danke ich Herrn Professor Dr. H.J. Schneider.

Obwohl meine Ergebnisse teilweise zu bekannten Arbeiten im Gegensatz stehen - stellvertretend seien R.J. Lipton, Peter Bathelt, Lothar Mackert genannt -, liegt es mir fern, diese irgendwie schmälern zu wollen; im Gegenteil wären meine Untersuchungen ohne die ihren undenkbar.

Den Herren Jäckel und Diller von der Firma Siemens (ZN Nürnberg) danke ich für die Möglichkeit der freien Mitarbeit in ihrer Abteilung, die mir noch Zeit ließ für die Promotion.

Mein Dank gilt nicht zuletzt Christel, meinen Eltern und allen Freunden, die mich in den Tagen der Ratlosigkeit ermutigt haben, weiterzumachen.



Digitized by the Internet Archive
in 2026

https://archive.org/details/IA41546221_0008

A b s t r a c t

WINKELMANN, KLAUS:

PROZESSKOORDINIERUNG DURCH VERWENDUNG VON FORTSETZUNGSBEDINGUNGEN

Um die komplexen Probleme der Wechselwirkung zwischen nebenläufigen Prozessen zu verstehen, braucht man ein formales Modell.

Bisherige Modelle (vgl. R.J. Lipton, Limitations of Synchronization Primitives, Research Report #31, Yale University, 1974) arbeiten mit unteilbaren Aktionen, deren Ausführung von einer Vorbedingung abhängig ist, in der Art

when Bedingung do Aktion;

Praktische Überlegungen und die Betrachtung von Implementierungen legen das umgekehrte Vorgehen nahe, nämlich unteilbare Aktionen, deren Ausführung erst mit Eintreten einer Fortsetzungsbedingung abgeschlossen ist, in der Art:

do Aktion wait Bedingung;

Nach diesem Konzept definieren wir ein Prozeßmodell. Die Eigenschaften solcher Prozeßsysteme werden anhand eines Simulationsbegriffs untersucht. Dieser wiederum baut auf dem Konzept des Schedules auf, das wir zu diesem Zweck formal beschreiben.

Es wird eine Reihe von Synchronisationsmechanismen definiert, die alle mit Fortsetzungsbedingungen arbeiten; diese werden miteinander verglichen und in eine Hierarchie eingeordnet. Solche Mechanismen enthalten gegenüber den bekannten mehr Strategie-Entscheidung; dadurch lassen sich Eigenschaften wie Fairness (Aushungerungs-Freiheit) leichter verwirklichen.

Es wird nachgewiesen, daß ein System mit Vorbedingungen nur unter starken Einschränkungen von einem solchen mit Fortsetzungsbedingungen ersetzt werden kann (und umgekehrt); d.h. bei strengen Anforderungen an den Simulationsbegriff (d.h. an die Portabilität) ergeben sich wesentliche Unterschiede zwischen den beiden Ansätzen.

Da wir andererseits nachweisen, daß die meisten real vorkommenden Implementierungen Fortsetzungsbedingungen verwenden, ist ein Systementwurf mit Vorbedingungen nicht ohne weiteres sinnvoll - er kann unter Umständen mit gegebenen Mitteln nicht (vollständig und sicher) realisiert werden.

Inhaltsverzeichnis

Einführung und Überblick	1
1. Bekannte Ansätze zur Prozeßkoordinierung	4
1.1. Das Modell von Lipton	4
1.2. Programmiersprachliche Konstrukte	8
2. Motivation für ein neues Modell	13
2.1 Der sleep/wakeup- Mechanismus in UNIX	13
2.2. Die Siemens-Implementierung von Semaphoren (K00R)	17
2.3. Das zweite Leser/Schreiber-Problem	23
2.4. Implementierung von P/V-multiple	26
2.5. Synchronisation und Scheduler	31
3. Entwicklung und Definition des Modells	34
3.1. Prozeßsystem	34
3.2. F- und V-Systeme	38
3.3. Schedules	42
3.4. Simulationsbegriff	46
3.5. Diskussion verschiedener Scheduleklassen	52
3.6. Fundamental- und Invarianztheorem	57
4. Anwendungen	64
4.1. Erzeuger-Verbraucher-Probleme	67
4.2. Ungepufferte Koordinierung	73
4.2.1. sleep/wakeup	73
4.2.2. Aktualisierungsprobleme	74
4.2.3. Erweiterungen	77
4.2.4. Ungepufferte Starts	80
4.2.5. Übersicht	82
4.3. Belegungsprobleme	83
4.3.1. Das Philosophenproblem	88
4.3.2. Korrektheit und Verhungerungsfreiheit	90
4.3.3. Grenzen der Anwendbarkeit	99
4.3.4. Erweiterungen	101
4.4. Das zweite Leser/Schreiber-Problem	105
4.5. Two-way-bounded-Mechanismen	109

4.6. Die Leistungshierarchie	111
4.6.1. ANFO/FREI-chunk stärker ANFO/FREI	116
4.6.2. ANFO/FREI-mult stärker als ANFO/FREI-chunk	116
4.6.3. sleep-chunk/wakeup und sleep-mult/wakeup sind unvergleichbar	119
4.6.4. ANFO/FREI und sleep/wakeup sind unvergleichbar	121
4.7. Zur Implementierung	123
 5. V - Systeme	 126
5.1. Neudefinition der Simulation	127
5.2. Blockadefreie Simulation	130
 6. Vergleich von V- und F-Systemen	 137
6.1. Zustandsübergänge	138
6.2. Zeitfreie Simulation	141
6.2.1. Voraussetzungen	141
6.2.2. V-System simuliert F-System	147
6.2.3. F-System simuliert V-System	156
6.3. Ein negatives Resultat	160
 7. Weitere Gedanken zur Synchronisation	 163
7.1. Prozeßwechselsperre	163
7.2. Prozeßkoordinierung in der Praxis	167
7.3. Modellierung echter Parallelität	172
 8. Zusammenfassung	 175
 Literaturverzeichnis	 180
 Abkürzungen und Symbole	 185

Einführung und Überblick

Auf fast allen Gebieten der Rechneranwendung treten parallele Abläufe auf, sei es in Betriebssystemen, Echtzeitsystemen, Rechnernetzen oder Datenbanken, sei es als Strukturierungsmittel oder zur Effizienzsteigerung.

Für die Koordinierung paralleler Prozesse gibt es die unterschiedlichsten Ansätze, von gemeinsamen Variablen über Semaphore, Monitore, Pfadausdrücke und andere bis zum Rendezvous-Konzept der Sprache ADA. Um die komplexen Probleme der Wechselwirkung zwischen nebenläufigen Prozessen zu bewältigen, empfiehlt sich - noch dringender als bei sequentiellen Aufgaben - ein systematisches Vorgehen bei Problembeschreibung, Spezifikation und Programmierung. Zuverlässige oder gar verifizierbare Programme können nur entstehen, wenn zwischen Spezifikation und Implementierung ein nachprüfbarer Zusammenhang besteht. Nun verwendet eine Implementierung üblicherweise betriebssystemnahe Funktionen, wie Blockieren oder Anhalten von Prozessen, während die Spezifikation schon der Verständlichkeit halber auf einem abstrakteren Niveau formuliert sein sollte; hierher gehören Eigenschaften wie Verklemmungsfreiheit, gegenseitiger Ausschluß oder Festlegung von Prioritäten.

Um überhaupt zu entscheiden, ob ein gegebenes Programm eine gegebene Aufgabenstellung löst, - und erst recht, um auf systematischem Wege eine Lösung zu finden - , bedarf es eines gemeinsamen formalen Modells, das die Semantik von Spezifikation und Implementierung festlegt.

Bei sequentiellen Aufgaben bieten sich oft Modelle so unmittelbar an, daß man sich ihrer gar nicht bewußt ist, etwa die mathematische Arithmetik als Modell für die integer- Arithmetik eines Computers; daß dieses Modell nur mit Einschränkungen zutrifft, soll hier nicht interessieren.

Die Modellierung paralleler Abläufe erweist sich als schwieriger, ist aber seit etwa einem Jahrzehnt Gegenstand der Forschung.

Bisherige Modelle (LIPT74) arbeiten mit unteilbaren Aktionen, deren Ausführung von einer Vorbedingung abhängig ist, in der Art

when Bedingung do Aktion;

In Abschnitt 1. stellen wir das Modell von Lipton vor, sowie unterschiedliche Sprachkonstrukte, die ebenfalls auf solchen Vorbedingungen beruhen.

Praktische Überlegungen und die Betrachtung von Implementierungen (Abschnitt 2.) zeigen, daß diese Ansätze das Verhalten gewisser real vorkommender Prozeßsysteme nicht befriedigend darstellen. Die Beispiele legen das umgekehrte Vorgehen nahe, nämlich unteilbare Aktionen, deren Ausführung erst mit Eintreten einer Fortsetzungsbedingung abgeschlossen ist, in der Art:

do Aktion wait Bedingung;

Wir bezeichnen Systeme mit Vorbedingungen als V-Systeme, solche mit Fortsetzungsbedingungen als F-Systeme.

In Abschnitt 3. definieren wir ein Prozeßmodell, das beide Arten als Spezialfälle umfaßt.

Die Eigenschaften solcher Prozeßsysteme untersuchen wir anhand eines Simulationsbegriffs, der auf dem Konzept des Schedules aufbaut. Zu diesem Zweck formalisieren und untersuchen wir erstmals ausführlich die Rolle von Schedules. Es stellt sich dabei heraus, daß unterschiedliche Voraussetzungen über die Zuteilungsstrategie des Betriebssystems zu unterschiedlichen Lösungsbegriffen führen.

Als Anwendungen definieren wir in Abschnitt 4. eine Reihe von Synchronisationsmechanismen, die alle mit Fortsetzungsbedingungen arbeiten; diese werden miteinander verglichen und in eine Hierarchie eingeordnet. Insbesondere die Forderung nach Fairness (Vermeiden von Aushungern) läßt sich mit diesen Systemen gut darstellen.

Als Spezialfall unseres Simulationsbegriffs stellt sich die blockadefreie Simulation dar, wie sie von Lipton bekannt ist (Abschnitt 5).

In Abschnitt 6. untersuchen wir, wieweit Systeme mit Vor- und solche mit Fortsetzungsbedingungen einander ersetzen können, bzw. in welchen Fällen die beiden Ansätze gleichwertig sind. Diese Frage ist von

Bedeutung für den häufigen Versuch, ein als V-System spezifiziertes Problem mit einem F-System zu lösen. Es ergibt sich, daß ein V-System nur unter starken Einschränkungen von einem F-System realisiert werden kann und umgekehrt.

Abschnitt 7. diskutiert die Grenzen unseres Modells und enthält weitere Gedanken zur Prozeßkoordinierung, die mit dem Modell nicht in direktem Zusammenhang stehen.

In Abschnitt 8. erörtern wir die Ergebnisse zusammenfassend, insbesondere unter dem Gesichtspunkt ihrer Bedeutung für die methodische Entwicklung asynchroner Programmsysteme.

1. Bekannte Ansätze zur Prozeßkoordinierung

1.1. Das Modell von Lipton

In dieser Arbeit beziehen wir uns immer wieder auf die Arbeit von Lipton (LIPT74), sowie deren Weiterführung durch Bathelt (BATH82). Es handelt sich dabei um ein streng formales Modell für parallele Prozeßsysteme; es wurde besonders im Hinblick auf den Vergleich von Synchronisationsmechanismen entwickelt. Die vorliegende Arbeit behandelt annähernd die gleichen Themen mit einem neuen Ansatz.

Wir folgen in diesem Abschnitt weitgehend der Darstellung von (BATH82) und stellen die wichtigsten Definitionen zusammen. Zitate aus (BATH82) sind in diesem Abschnitt nicht ausdrücklich gekennzeichnet.

Prozeßsystem

$P = (A, D, w)$ heißt Prozeßsystem, wenn gilt:

(1) $D = D_1 \times \dots \times D_n \times D_g$ ist endliches kartesisches Produkt mit typischem Element $(l_1, \dots, l_n, G)$ und heißt Zustandsmenge, $w \in D$ ist der Anfangszustand und A ist eine Aktionenmenge.

Es gibt zwei Funktionen

$$\text{adr}: A \rightarrow D_1 \cup \dots \cup D_n \quad \text{und} \quad \text{bz}: A \rightarrow \{1, \dots, n\}$$

mit den Eigenschaften:

(2) Eine Aktion $f \in A$ ist eine Abbildung von D nach D , definiert in der Art

$$\text{when } L(\text{bz}(f)) = \text{adr}(f) \quad \text{and } p(G)$$

$$\text{do } L(\text{bz}(f)) := s(G); G := t(G);$$

(3) Für zwei Aktionen $f, g \in A$ gilt:

$$\text{adr}(f) = \text{adr}(g) \quad \text{and} \quad \text{bz}(f) = \text{bz}(g) \Rightarrow f = g,$$

d.h. besitzen die Aktionen f und g die gleiche Adresse und gehören sie zum gleichen Prozeß, dann sind sie identisch.

Ein Prozeß P_i wird dabei angesehen als die Menge der Aktionen f mit

$$\text{bz}(f) = i.$$

Eine endliche Folge von Aktionen aus A heißt Ablauf, dessen Wirkung auf

den Systemzustand wird in natürlicher Weise definiert, ebenso die Funktion lf , die jedem Ablauf α die Menge derjenigen Aktionen zuordnet, deren Ausführungsbedingung nach der Berechnung α in P erfüllt ist, d.h. die nach Ablauf von α lauffähig sind.

Ein Ablauf

$$\alpha = a_1 a_2 \dots a_n$$

heißt dann aktiv, wenn für jedes $i \in \{1, \dots, n\}$ gilt:

$$a_i \in lf(a_1 a_2 \dots a_{i-1}).$$

Der Begriff der Blockierung wird in dieser Definition nicht gebraucht, läßt sich aber leicht darstellen:

Ein Prozeß P_i heißt blockiert nach dem Ablauf α , wenn

$$lf(\alpha) \cap P_i = \emptyset,$$

d.h. wenn keine seiner Aktionen ausführbar ist.

Diese allgemeine Form von Prozeßsystemen ist für praktische Untersuchungen wenig geeignet; man fordert daher zusätzlich eine Trennung zwischen synchronisierenden und anderen Aktionen und beschränkt zugleich die Synchronisation auf ganzzahlige Variable. Dies geschieht im Begriff

Synchronisationssystem:

Ein Prozeßsystem $P = (A, D, w)$ mit der Zustandsmenge

$$D = D_1 \times \dots \times D_n \times (Z^k \times E)$$

und dem typischen Element $(L_1, \dots, L_n, (S_1, \dots, S_m), H)$ heißt Γ -Synchronisationssystem, wenn jede Aktion eine der beiden folgenden Formen besitzt;

(1) when $L_i = a$ do $(L_i, H) := (b(H), h(H))$;

(2) when $L_i = a$ and $V(S_1, \dots, S_m)$ do $L_i := a'$; $A(S_1, \dots, S_m)$;

mit $b(H) \neq a$, $a' \neq a$. $(L_i, H) := (b(H), h(H))$ ist eine parallele Wertzuweisung.

Aktionen des Typs (1) werden auch als nichtsynchronisierende und solche vom Typ (2) als synchronisierende Aktionen bezeichnet.

$V(S_1, \dots, S_m)$ ist die Synchronisationsbedingung oder (in dieser Arbeit) Vorbedingung und $A(S_1, \dots, S_m)$ die Synchronisationsoperation einer Aktion. Die S_i werden auch als Semaphorvariable bezeichnet. Ihr Wertebereich ist die Menge Z der ganzen Zahlen. Γ ist der System-

name. Bathelt definiert über 40 Systemtypen jeweils durch Angabe der Paare (V,A) , wie etwa

(D4): P heißt PV-System, wenn für jede Synchronisationsaktion von P entweder (1) oder (2) gilt:

- (1) $V ::= S_j > 0; \quad A ::= S_j := S_j - 1;$
- (2) $V ::= \text{true}; \quad A ::= S_j := S_j + 1;$

Um mit diesem Modell Synchronisationsmechanismen miteinander zu vergleichen und Fragen zu beantworten wie "löst ein gegebenes Prozeßsystem eine gegebene Synchronisationsaufgabe?", definiert Lipton einen

Simulationsbegriff:

Seien $P=(A1,D1,w1)$ und $Q=(A2,D2,w2)$ Prozeßsysteme. Eine Abbildung

$r: A2^* \rightarrow A1^*$ heißt Realisierung von P durch Q, wenn r ein Halbgruppenhomomorphismus ist.

(A^* bezeichnet das freie Monoid über dem Zeichenvorrat A, mit der Einheit $\lambda =$ leere Folge).

Eine Aktion a heißt Buchführungsaktion, wenn $r(a)=\lambda$, beobachtbar sonst.

Q simuliert P durch r, wenn r eine Realisierung von P durch Q mit den folgenden Eigenschaften ist:

(1) r ist vollständig und sicher, d.h.

$$r \{ \alpha: \alpha \text{ aktiv in } Q \} = \{ \beta: \beta \text{ aktiv in } P \}.$$

r ist vollständig bedeutet:

$$r \{ \alpha: \alpha \text{ aktiv in } Q \} \supseteq \{ \beta: \beta \text{ aktiv in } P \}.$$

r ist sicher bedeutet:

$$r \{ \alpha: \alpha \text{ aktiv in } Q \} \subseteq \{ \beta: \beta \text{ aktiv in } P \}.$$

(2) r ist blockadefrei, d.h.

$$1f(\alpha) \cap Q_i = 0 \Rightarrow 1f(r(\alpha)) \cap r(Q_i) = 0, \text{ wobei } \alpha \in A2^*.$$

(3) r ist prozeßtreu, d.h.

für zwei beobachtbare Aktionen $f, g \in A_2$ muß gelten:

(a) $bz(f) = bz(g) \Rightarrow bz(r(f_i)) = bz(r(g))$,

(b) $r(f) = r(g) \Rightarrow bz(f) = bz(g)$.

Wir sagen: Q simuliert P durch r prompt, wenn r neben (1)-(3) noch die Eigenschaft (4) besitzt:

(4) r ist prompt, d.h.

es existiert eine Konstante $c > 0$, so daß für alle aktiven Abläufe α von Q gilt: $1 + \#r(\alpha) \geq c \times \#\alpha$.

Dabei ist $\#$ der Operator, der die Länge (Anzahl der Glieder) einer Folge angibt.

Mit diesen Begriffen konnte nun Lipton Unterschiede zwischen Synchronisationsmechanismen nachweisen in dem Sinn "es gibt ein Prozeßsystem vom Typ T_1 , das nicht von einem Prozeßsystem vom Typ T_2 simuliert werden kann". Er ordnete die bekanntesten Mechanismen, wie verschiedene Semaphoreoperationen und verschiedene Typen von Petri-Netzen in eine Leistungshierarchie ein; diese wurde von Bathelt wesentlich erweitert. Mit Hilfe zusätzlicher Vergleichskriterien, auf die wir hier nicht eingehen, gelang es Bathelt, sehr feine Unterschiede zwischen Synchronisationsmechanismen nachzuweisen; daß ein Mechanismus einen anderen voll ersetzen kann, ist demnach fast nur noch in den Fällen möglich, wo der den anderen bereits aufgrund der Definitionen umfaßt.

1.2. Programmiersprachliche Konstrukte

In diesem Abschnitt stellen wir verschiedene Konzepte zur Koordinierung paralleler Prozesse vor, die mehr als das Modell des vorigen Abschnitts auf die Anwendung hin ausgerichtet sind. Wir folgen dabei im wesentlichen der Darstellung in (HOFM82).

Guarded Commands

Die von Dijkstra (DIJK68, DIJK75) präsentierte Idee von "guarded commands" wurde von C.A.R. Hoare (HOAR78) zu einem ausdrucksstarken und zugleich einfachen Konzept "kommunizierender sequentieller Prozesse" ausgebaut. Hoare definiert eine Sprache, deren wesentlichen Elemente guarded commands und input/output commands sind:

Ein guarded command ist von der Form

$(\text{guarded command}) ::= (\text{guard}) \rightarrow (\text{command list}).$

Dabei ist (guard) im einfachsten Fall eine logische Bedingung; aus guarded commands können Alternativen oder Iterationen aufgebaut werden:

$(\text{alternative command}) ::= [(\text{guarded command}) \{ \alpha(\text{guarded command}) \}]$

$(\text{repetitive command}) ::= * (\text{alternative command}).$

Die Interpretation besagt dabei: falls einer der guards erfüllt ist, wird eine derjenigen command lists ausgeführt, deren guard erfüllt ist. Andernfalls bleibt die Alternative wirkungslos, die Iteration wird abgebrochen.

Input/output commands dienen zur Kommunikation der Prozesse untereinander und sind auch das einzige Mittel der Kommunikation und Koordinierung; sie haben die Form:

$(\text{input command}) ::= (\text{source}) ? (\text{target variable})$

$(\text{output command}) ::= (\text{destination}) ! (\text{expression})$

und die Interpretation:

(source) bzw. (destination) sind die Prozeßnamen von Quelle bzw. Empfänger; der Empfänger wird beim input command solange blockiert, bis der Sender ein passendes output command durchlaufen hat; dann wird der Wert des gesendeten Ausdrucks in die Zielvariable übertragen. Der Sender wird nicht blockiert.

Vielfältige Anwendungen werden ermöglicht durch die Zulassung von input commands als guards:

```
(guard) ::= (guard list)
          | (guard list) ; (input command)
          | (input command)
```

wobei (guard list) eine Liste logischer Ausdrücke darstellt.

Die Interpretation liegt nahe: so bedeutet die Alternative

```
[X ? V() → val := val +1
  □ val > 0; Y ? P() → val := val -1]
```

folgendes: falls der Prozeß X an den vorliegenden Prozeß Z den Aufruf "Z ! V()" sendet, wird die Variable val erhöht. Falls sie größer als 0 ist, besteht außerdem Empfangsbereitschaft für den Aufruf P() vom Prozeß Y; er führt zum Erniedrigen der Variablen.

Dieses Beispiel kann Teil der Realisierung eines Semaphors sein, allerdings müssen dabei als Prozesse X und Y alle potentiellen Benutzer explizit benannt werden.

Hoare gibt mit dieser Methode ein elegantes Programm zur Matrizenmultiplikation an.

Da die Methode in Hinsicht auf die Prozeßkommunikation nur auf einem einzigen Paar von Grundoperationen (Senden, Empfangen) beruht, ist sie modellhaft für Nachrichtensysteme, wie sie auf manchen Rechnern existieren.

Ein vergleichbares Konzept von P.B. Hansen (HANS78) benutzt außer guarded commands sogenannte guarded regions

```
when( guard1 → command1 □ ... □ guardn → commandn)
```

die sich von den guarded commands dadurch unterscheiden, daß die Anweisung blockiert wird, falls kein guard erfüllt ist.

Conditional critical regions

S. Owicki und D. Gries (OWIC76) haben ein Konzept der Parallelarbeit entworfen, das insbesondere auf die Verifizierbarkeit der entstehenden Programme hin angelegt ist. Neben den üblichen Anweisungen und einem cobegin ... coend - Statement

für Parallelanweisungen kennt ihre Sprache die Konstruktion

```
with r when B do S
```

zur Formulierung kritischer Abschnitte.

Dabei bedeutet r eine (als "resource" bezeichnete) Menge gemeinsamer Variablen - solche dürfen nur in kritischen Abschnitten benutzt werden; B ist ein logischer Ausdruck und S eine Anweisung, in der r vorkommen kann.

"Wenn ein Prozeß diese Anweisung auszuführen versucht, wird er verzögert, bis die Bedingung B wahr ist und r nicht von einem anderen Prozeß benutzt wird. Wenn der Prozeß zu r Zugang (control) hat und B wahr ist, wird S ausgeführt" (wörtlich übersetzt aus (OWIC76)).

Für diese Anweisungen geben die Autoren formale Verifikationsregeln nach der bekannten Methode der Zusicherungen von Hoare an.

Prozeßkommunikation in ADA

Das wesentliche Kommunikationsmittel in ADA sind entries (Eingänge). Sie werden mit der Anweisung "accept" notiert und können außerdem von einer Vorbedingung abhängen, sowie zu Alternativen zusammengefaßt werden, so daß sich folgende allgemeine Form ergibt (selective wait, nach (PYLE82)):

```

when Cond0
select  when Cond1 accept Entry1(Param1) do Stmt1A end Stmt1B
or      when Cond2 accept Entry2(Param1) do Stmt2A end Stmt2B
or      ...
else    StmtX
end select

```

Dabei bedeuten Cond0, Cond1 ... logische Ausdrücke,
 Entry1, Entry2 ... die Namen der Eingänge,
 Param1, Param2 ... die formalen Parameter,
 Stmt1A, Stmt1B ... Anweisungsfolgen.

Ein Eingang kann von anderen Prozessen durch Nennung des Namens und Versorgung der Parameter aufgerufen werden. In diesem Fall wird ein "Rendezvous" zwischen den beiden Prozessen hergestellt; d.h. der Prozeß, welcher zuerst bereit ist zur Kommunikation, wartet auf den anderen. Die zu dem Eingang gehörige, mit "do ... end" geklammerte

Anweisungsfolge Stmt1A läuft direkt während des Rendezvous, also unter gegenseitigem Ausschluß mit dem aufrufenden Prozeß ab, anschließend kann eine weitere Anweisungsfolge Stmt1B parallel bearbeitet werden.

Das select-Statement ist dem guarded command von Hoare verwandt, aber erheblich raffinierter, und weicht auch in anderen Punkten ab:

1. der Sender wird beim "accept" nicht genannt.
2. bezüglich Blockierungen besteht eine gewisse Symmetrie, anders als bei Hoare, wo der Sender sich nicht blockiert.
3. Die Zuteilung der Anweisungsfolge in exklusive und parallele ist ebenfalls neu.

Die Betriebssystemmaschine

S. Keramidis und L. Mackert (KEMA79,KERA82,MACK83) definieren eine Sprache zur Beschreibung asynchroner Systeme, die auf abstrakten Datentypen basiert. Die Methode eignet sich gleichermaßen zur Spezifikation, Implementierung und Verifikation.

Zur Definition eines abstrakten Datentyps gehört neben der Festlegung der abstrakten Datenstruktur und der Wirkung der einzelnen Operationen ein Synchronisationsanteil; die Synchronisation findet also nicht in den aufrufenden Prozessen, sondern am gemeinsam benutzten Objekt selbst statt. Die Synchronisation wird dabei bestimmt durch

- Angabe von Verträglichkeiten zwischen verschiedenen Aktivitäten,
- Angabe von Prioritätsverhältnissen und
- Nicht-Blockierungsbedingungen, die jeder einzelnen Aktivität vorgeschaltet sein können.

Letztere seien am Beispiel der "write"-Operation auf einem beschränkten Puffer demonstriert:

```
write (i: inf);
    PRE:      ...
    NBL:      belegt < max;
    EFFECTS:  ...
```

Das Schlüsselwort "NBL" leitet eine Bedingung ein, die erfüllt sein muß, damit die Ausführung der Operation nicht blockiert ist /zitiert

nach (MACK83), S.39).

"PRE" ist eine Bedingung, die beim Aufruf der Operation zugesichert wird, und "EFFECTS" beschreibt deren (sequentielle) Wirkung.

In der erweiterten Form dieser Methode können zu einer (nach außen sichtbaren) Operation mehrere Paare von NBL- und EFFECTS-Prädikaten gehören. Die Interpretation verlangt jedoch immer, daß zum Zeitpunkt der Ausführung einer (Teil-) Aktivität die zugehörige Vorbedingung "NBL: ..." gilt.

Zusammenfassung

Die hier kurz vorgestellten Konzepte zur Koordinierung paralleler Prozeßsysteme unterscheiden sich hinsichtlich Zielsetzung, Methode, Komplexität und Anwendungsbereich.

Dennoch haben alle eines gemeinsam, nämlich die Vorstellung einer Vorbedingung, an der sich ein Prozeß blockieren kann. Diese Vorstellung steht im Einklang mit der Formulierung einer elementaren Aktion

when B do A

im Modell von Lipton.

Alle diese Vorbedingungen / guards / NBL-Bedingungen werden also so verstanden: es genügt für das Fortschreiten des blockierten Prozesses nicht, daß die Bedingung irgendwann gegolten hat - vielmehr muß sie bei der Ausführung der anstehenden Aktion gelten.

2. Motivation für ein neues Modell

In diesem Abschnitt soll anhand einiger Beispiele die Notwendigkeit für ein neues Modell für Synchronisationsmechanismen aufgezeigt werden. Die Beispiele gehen von existierenden Implementierungen aus; es wird argumentiert, daß Lipton's Modell deren Eigenschaften nicht ausreichend widerspiegelt.

Gewicht haben diese Beispiele dadurch, daß es sich nicht um exotische Ausnahmen handelt; vielmehr funktionieren die meisten oder alle Implementierungen nach ähnlichen Prinzipien.

Auf dem Hintergrund der Beispiele werden dann informal die Ideen entwickelt, die dem im nächsten Kapitel präsentierten Modell zugrundeliegen.

2.1. Der sleep/wakeup - Mechanismus in UNIX

Das Betriebssystem UNIX(*) ist weltweit eines der meistverbreiteten und wird nach Expertenmeinung auch in Zukunft an Bedeutung gewinnen. Die Untersuchung der Synchronisations-Primitive in UNIX war der ursprüngliche Anstoß zur vorliegenden Arbeit.

Die in Frage stehenden Operationen lassen sich informal so beschreiben:

```
sleep(e)  :   der anfordernde Prozeß wird blockiert und als
               "wartend auf das Ereignis e" vermerkt.

wakeup(e) :   alle auf das Ereignis e wartenden Prozesse werden
               deblockiert. Der aufrufende Prozeß bleibt ebenfalls
               ablauffähig.
```

Dem an Semaphore gewöhnten Programmierer fällt dabei zunächst auf:

- diese "Ereignisse" werden nicht gepuffert, d.h. ein "sleep" blockiert sich immer, auch wenn vorher schon "wakeup" durchlaufen wurde;
- ein "wakeup" kann andererseits beliebig viele schlafende Prozesse aufwecken.

(*) UNIX is a Trademark of Bell Laboratories

Es sind jedoch nicht diese Eigenschaften, die eine Modellierung im Stile Lipton's problematisch machen, sondern eine etwas subtilere:

Der Prozeß, der "sleep" aufruft, darf diese Operation in jedem Fall ausführen; erst dann wird er blockiert und wartet auf Fortsetzung.

Der Versuch einer Modellierung zeigt dies klar:

```
sleep(e) -  s11: W(e) := W(e) U {ownpid} ;
            s12: when ownpid ∈ W(e) do skip;
wakeup(e) -  wu: W(e) := 0;
```

Dabei bedeuten:

```
ownpid    -  eigene Prozeßidentifikation (Prozeßnummer) des
              aufrufenden Prozesses,
W(e)      -  eine dem Ereignis e zugeordnete Menge wartender
              Prozesse,
skip      -  leere Operation.
```

Der entscheidende Punkt ist, daß hier die sleep-Operation in zwei Aktionen aufgeteilt wurde - eine Anmeldeoperation und das Warten auf Fortsetzung. Eine solche Zweiteilung ist notwendig, weil "sleep" immer ausführbar sein muß, aber zu einer Blockade führt - im Lipton-Modell treten Blockaden nur jeweils vor einer Operation auf.

Daß eine solche Aufteilung von synchronisierenden Operationen zu Problemen führen kann, ist bekannt. Diese entstehen im allgemeinen bei einer Unterbrechung innerhalb der Operation; es stellt sich also die Frage, ob ein Prozeßsystem mit obigen Operationen wirklich das gleiche Verhalten zeigt wie das gleiche System bei Verwendung von sleep/wakeup. Die Präzisierung dieses "Gleichverhaltens", nämlich ein Simulationsbegriff, wird erst später geschaffen, hier sollen nur informelle Überlegungen stehen.

Man betrachte also folgende Operationsfolge in einem System aus zwei Prozessen Q1, Q2:

```
(Q1: s11) (Q2: wu) (Q1: s12)
```

(Die Schreibweise "(nn: aaa)" soll von hier ab bedeuten : der Prozeß nnn führt die Aktion aaa aus. Da sich in diesem Beispiel alle Operationen auf das gleiche Ereignis e beziehen, wird "e" weggelassen).

Die entsprechende Sequenz im sleep/wakeup-System wäre:

(P1: sleep) (P2: wakeup)

D.h. die Aktion sl2 ist unbeobachtbar. Wie sehen nun die Blockaden in den beiden Fällen aus ?

Sicher ist Prozeß 2 nirgends blockiert. Prozeß P1 ist zweifellos nach sleep blockiert; aber ist es auch Prozeß Q1 nach der Teiloperation sl1? Es gibt hierauf zwei mögliche Antworten:

- im Lipton-Sinne ist dieser Prozeß blockiert, weil er keine Aktion ausführen kann; d.h. die Vorbedingung der als nächste anstehenden Operation ist nicht erfüllt.
- Aus der Sicht des Betriebssystems ist der Prozeß nicht blockiert, denn er wurde unterbrochen, bevor er sl2 aufrufen konnte.

Wenn man nicht zuläßt, daß das Betriebssystem im voraus die Befehle der Anwenderprozesse analysiert, kann es auch nicht wissen, wann sich ein solcher blockieren wird.

Wenn man sich für die zweite Antwort entscheidet, die weiter unten noch ausführlich diskutiert werden wird, dann folgt daraus: das Prozeßsystem Q mit sl1/sl2 zeigt ein anderes Verhalten als P, genauer: es gibt Fälle, wo Q ohne Blockaden auskommt, P nicht.

Da das Betriebssystem die Information, welche Prozesse bisher wie oft blockiert waren, bei der weiteren Prozessorvergabe verwerten könnte, folgt daraus ein möglicherweise anderes Verhalten. Da dies im nächsten Beispiel deutlicher zu sehen sein wird, wird es hier nicht weiter ausgeführt.

Ein anderer Gesichtspunkt in diesem Beispiel ist die Komplexität der Darstellung: Obwohl der sleep/wakeup-Mechanismus intuitiv als sehr elementar angesehen werden darf - gewiß von keinem höheren Niveau als etwa Semaphore - benötigt die obige Darstellung durch sl1/sl2 recht komplexe Daten, nämlich Mengen von Prozessen, statt etwa integers. Folgende Formulierung zeigt, daß es bei Verwendung von Fortsetzungsbedingungen anstelle von Vorbedingungen auch anders geht:

```
sleep(e) - do b(e) := 1 wait b(e) = 0;
wakeup(e) - do b(e) := 0;
```

Hier ist $b(e)$ eine dem Ereignis e zugeordnete binäre Variable und die Klausel

do A wait B ;

bezeichnet eine elementare Operation, die so zu verstehen ist:

Die Aktion aaa wird zunächst unbedingt ausgeführt, anschließend wartet der Prozeß auf Eintreten der Bedingung bbb . Sobald bbb gilt, ist der Prozeß deblockiert und diese Operation abgeschlossen.

Die do-wait-Klausel - eines der Kernstücke dieser Arbeit - wird später (3.2.) formal eingeführt.

Als Essenz diese Abschnitts bleibt festzuhalten:

a) die Darstellung mit elementaren Aktionen der Art "when bbb do aaa ;" beschreibt nicht exakt das Verhalten des sleep/wakeup-Mechanismus in Bezug auf Blockierungen.

b) diese Darstellung ist komplexer, als es dem Mechanismus angemessen erscheint.

2.2. Die Siemens-Implementierung von Semaphoren

Dieses Beispiel geht aus vom Betriebssystem ORG 300 PV für die Siemens-Prozeßrechner der Serie 300 /R30. Die folgenden Beobachtungen dürften aber für die meisten Implementierungen gelten.

Semaphore heißen im ORG 300 PV Koordinierungszähler; auf solchen gibt es zwei Operationen K00R-(s) und K00R+(s), die den bekannten Operationen P(s) und V(s) entsprechen und wie folgt definiert sind.

```

K00R-(s)  :  s.ct := s.ct - 1;
              if s.ct < 0
              then begin enqueue(s.q, ownpid);
                     blockiere (ownpid);
              end;

K00R+(s)  :  s.ct := s.ct + 1;
              if s.ct ≤ 0
              then begin deblockiere (first(s.q));
                     s.q := tail(s.q);
              end;

```

Hierbei stehen

s.ct für die mit dem Semaphor assoziierte Zählvariable,
s.q für die zugehörige Warteschlange von Prozeßnummern,
first bezeichnet das erste Element der Warteschlange,
tail die Warteschlange ohne erstes Element,
enqueue steht für Einfügen in die Warteschlange, wobei hier offen-
gelassen wird, ob dies nach FIFO oder einer anderen
Strategie geschieht (Im ORG300PV wird das beim Einrichten
des Koordinierungszählers festgelegt).

Die Operationen blockiere/deblockiere haben die naheliegende Bedeutung.

Diese Beschreibung läßt ein wichtiges Merkmal von K00R+/- bewußt beiseite; In Wirklichkeit handelt es sich nämlich um two-way-bounded-Operationen, d.h. es gibt eine Obergrenze, die die gleiche Rolle spielt wie die Untergrenze 0, so daß zwischen den Operationen K00R+ und K00R- eine Symmetrie besteht. Wir nehmen jedoch vereinfachend die Obergrenze so groß an, daß sie nie erreicht wird, und dann trifft die obige

Beschreibung zu.

Weitere Möglichkeiten von K00R+/- lassen wir ebenfalls außer Betracht, der Vollständigkeit halber seien sie kurz erwähnt:

Die Aufrufe können mit einer Zeitmodifikation ("nach 2 Sekunden", "alle 20 Minuten" etc.) versehen werden und können Daten senden und empfangen.

Die beschriebene Grundversion der Koordinierungsaufrufe hält man nun allgemein für eine Implementierung der bekannten Semaphore-Operationen; deren Definition sei hier nochmals vergegenwärtigt:

$P(s) : \text{ when } s > 0 \text{ do } s := s - 1;$

$V(s) : s := s + 1;$

Daß hier die Zählvariable keine negativen Werte annimmt, was sie bei K00R+/- durchaus tun kann, ist gewiß nur eine oberflächliche Feststellung. Wichtiger ist:

bei K00R+/- kann - wie bei sleep/wakeup - jede Operation, wenn sie aufgerufen wird, zunächst "etwas tun", d.h. eine Zustandsänderung herbeiführen, bevor sie blockiert wird; für die P-Operation gilt dies nicht. Nun könnte man auch dies noch für eine Spitzfindigkeit halten, aber die folgenden Überlegungen mögen zeigen, wie unterschiedliches Verhalten von Prozeßsystemen daraus resultiert.

Nehmen wir an, das Betriebssystem realisiere einen einfachen nicht verdrängenden Schedule, d.h. es gehe bei der Prozessorzuteilung wie folgt vor:

1. wähle einen nicht blockierten Prozeß i aus und teile ihm den Prozessor zu.
2. Solange Prozeß i lauffähig ist, wird er nicht verdrängt. Erst wenn er blockiert wird, gehe nach 1 !

Betrachten wir dann folgendes Prozeßsystem:

semaphore s init 0;

process P1: loop V(s) ; P(s) endloop;

process P2: loop P(s) ; V(s) endloop;

Man sieht leicht, daß hier - obigen Schedule vorausgesetzt - der Prozeß P2 nie zum Zuge kommt, weil er anfangs blockiert ist, wogegen P1 nie blockiert ist.

Genauer gibt es zwei Betrachtungsweisen, die aber zum gleichen Ergebnis führen:

a) Im Lipton'schen Sinne ist P2 anfangs blockiert. Daher kann nur P1 laufen, dieser gibt den Prozessor nie auf.

b) wie im vorigen Abschnitt kann man annehmen, es gebe keine "vorausschauenden" Blockaden; P2 kann also zunächst den Prozessor zugeteilt bekommen; er blockiert sich dann beim "Aufruf" der P-Operation; von da an ist nur P1 aktiv.

Beachte, daß der Prozeß P2 durchaus immer wieder lauffähig wird - dies wirkt sich aber nicht aus, weil sein Konkurrent P1 immer lauffähig ist.

Wie verhält sich nun dieses Prozeßsystem, wenn man P/V durch K00R-/+ ersetzt ?

```
semaphore s init 0;
process Q1: loop K00R+(s) ; K00R-(s) endloop;
process Q2: loop K00R-(s) ; K00R+(s) endloop;
```

Anfangs ist kein Prozeß blockiert, und man muß zwei Fälle unterscheiden:

Fall 1:

als erster werde Q1 aktiviert. Dann läuft offensichtlich Q1 immer weiter, da es sich nie blockiert - also ein Verhalten ganz wie bei den Prozessen P1, P2.

Fall 2:

als erstes werde Q2 aktiviert. - Dann sieht alles ganz anders aus: es ergibt sich folgender Ablauf:

Prozeß /	Aktion /	s.ct	/ Zustand Q1 / Q2	
Q2	KOOR-	-1	n	b
Q1	KOOR+	0	n	n
Q1	KOOR-	-1	b	n
Q2	KOOR+	0	n	n
Q2	KOOR-	-1	n	b
Q1	KOOR+	0	n	n
Q1	KOOR-	-1	b	n
Q2	KOOR+	0	n	n

..... usw.

(n, b heißt nicht blockiert, blockiert).

Kurz: beide Prozesse wechseln sich dauernd ab.

Wenn man sich vor Augen führt, welche Zeit u. U. ein Prozeßwechsel kosten kann, insbesondere wenn nicht beide Prozesse gleichzeitig im Hauptspeicher präsent sein können, muß man feststellen:

Die Prozeßsysteme P und Q unterscheiden sich drastisch, zumindest in Bezug auf den angegebenen Schedule .

Man mag einwenden, daß es sich hier um keinen "fairen" Schedule handelt, weil Prozeß P2 ausgehungert wird; dies hat jedoch nichts mit dem eigentlichen Problem zu tun: ein etwas komplizierterer Schedule, der jeweils vorübergehend nach obiger Regel funktioniert, aber gelegentlich (etwa bei Zeitscheibenende !) doch einen Prozeß verdrängt, führt zur gleichen Diskrepanz zwischen P und Q .

Das Beispiel ist um so gewichtiger, da eine solche Schedule-Strategie nichts künstliches hat, sondern geradezu typisch ist für reale Betriebssysteme - angenommen wird ja nur, daß der Schedule nicht zu oft lauffähige Prozesse verdrängt.

Ebenfalls nicht von Bedeutung ist in diesem Beispiel die Strategie, nach der die Koordinierungszählerwarteschlange bearbeitet wird: da zu jedem Zeitpunkt höchstens ein Prozeß wartend ist, spielt es keine Rolle, ob diese Warteschlange nach FIFO oder nach einer Prioritätsstrategie bedient wird.

Da also die beobachtete Diskrepanz nicht aus einem exotischen Schedule oder anderen Besonderheiten resultiert, worin hat sie dann ihre Gründe?

a) ein K00R- wird im Gegensatz zur P-Operation schon vor der Blockade ausgeführt.

b) Daher ist es möglich, daß K00R- schon einen Effekt hat, auch wenn es sich anschließend blockiert. Dieser Effekt ist das "Anmelden" für den Semaphor. In der Weise des obigen Prozeßsystems werden ja Semaphore zur Belegung einfacher Betriebsmittel benutzt; aus dieser Sicht gilt für K00R+/- : sobald das Betriebsmittel frei wird (K00R+), wird es einem evtl. angemeldeten Prozeß zugeteilt. Genau das gilt für P/V nicht, weil eben das "Anmelden" dort nicht explizit betrachtet wird.

c) Somit ist in der Implementierung von K00R+/- schon die wesentliche Entscheidung über die Zuteilstrategie vorweggenommen (nämlich in der Prozedur enqueue). Bei P/V ist diese Entscheidung offengelassen; daher kann der Prozeß P1 seine P-Operation ausführen, wenn diese ansteht: das "Betriebsmittel" wurde noch nicht einem anderen Prozeß zugeteilt !

Entsprechendes ist in Q1 nicht möglich: da Q2 schon seinen Bedarf an s angemeldet hat, wurde es bei der Aktion

(Q1 : K00R+ (s))

deblockiert und hat damit den kritischen Abschnitt bereits betreten; Q1 muß sich also blockieren.

Man erkennt den entscheidenden Unterschied auch an der Definition von K00R+: dort wird nämlich ein Prozeß sofort deblockiert, falls einer wartet.

Im P/V- System gilt dagegen:

Wenn P2 an der P-Operation wartet, und P1 führt eine V-Operation aus, dann wird zwar P2 ebenfalls deblockiert, aber diese Deblockierung ist reversibel, d.h. falls der Prozeß P2 nicht aktiv wird, wird er wieder blockiert, sobald $s < 0$ gilt.

(Zu diesen Zustandsübergängen siehe auch Abschnitt 6.1.).

Nach all dem scheint es gerechtfertigt, zu behaupten:

KOOR+/- ist keine korrekte Implementierung von P/V, wenn P/V wie oben (d.h. wie üblich) definiert wird.

Es kann aber nicht Aufgabe dieser Arbeit sein, einen zufälligen Fehler in irgendeiner Implementierung nachzuweisen. Vielmehr deutet alles darauf hin, daß jede Implementierung auf einem Prinzip basiert, das nicht verträglich ist mit der obigen, üblichen Modellierung von P/V. Dieses Prinzip heißt:

Eine, wenn auch als unteilbar angesehene, Synchronisationsoperation kann jederzeit aufgerufen werden, und hat zum Aufrufzeitpunkt selbst dann einen Effekt, wenn sie dabei blockiert, also nicht vollständig ausgeführt wird.

Die Unterscheidung zwischen verschiedenen Zeitpunkten einer Operation ist freilich nicht neu, vgl. (HABE72), (MACK83).

Dabei ist jedoch nie der Gedanke aufgetaucht, daß man mit einem solchen Konzept eine ganz andere Klasse von Synchronisationsmechanismen beschreibt als mit dem Ansatz von unteilbaren Operationen der Art

```
when B do A ;
```

Üblicherweise werden abstrakte Systeme dieser Art (V-Systeme) implementiert durch Systeme mit expliziten Aufruf- und Ausführungszeitpunkten (F-Systeme), und zwar sowohl in praxi als auch in den genannten theoretischen Arbeiten. Ob und in welchem Sinne eine solche Implementierung überhaupt möglich ist, ist das Thema von Abschnitt 6.

2.3. Das zweite Leser/Schreiber-Problem

In diesem Abschnitt soll an einem Beispiel aus der Literatur aufgezeigt werden, warum ein Modell für Synchronisationsmechanismen explizit den Anfangs- (oder Aufruf-) Zeitpunkt einer Operation erfassen muß.

Als zweites Leser/Schreiber-Problem wird folgendes Synchronisationsproblem bezeichnet (HOFM82):

Eine unbekannte Anzahl von Prozessen benutzen eine gemeinsame Datenbasis; es werden lesende und schreibende Aufrufe unterschieden. Beliebige viele Leseaufrufe dürfen gleichzeitig aktiv sein, jedoch schließt ein Schreibaufruf jeden anderen gleichzeitigen Aufruf aus, d.h. Schreiber sind sowohl untereinander als auch mit den Lesern unverträglich. Dabei sollen (dies unterscheidet das zweite vom ersten Leser-Schreiber-Problem) die Schreiber Priorität vor den Lesern haben, d.h. sobald ein Schreibaufruf ansteht, darf kein weiterer Leser zugelassen werden.

Diese Aufgabe kann nach (LIPT74) als sogenanntes Up/Down-System dargestellt werden. Ein Up/Down-System hat folgende Primitive:

$$\{S_1, \dots, S_k\} : \text{up}(S_j), \text{ definiert als}$$

$$\text{when } S_1 + \dots + S_k \geq 0 \text{ do } S_j := S_j + 1;$$

und

$$\{S_1, \dots, S_k\} : \text{down}(S_j), \text{ definiert als}$$

$$\text{when } S_1 + \dots + S_k \geq 0 \text{ do } S_j := S_j - 1;$$

wobei $S_1, \dots, S_k, S_j$ durch integers repräsentierte Semaphore darstellen.

Damit läßt sich die Aufgabe wie folgt darstellen (nach (LIPT74), zitiert nach (BATH82)):

Semaphores S1,S2,S3; init S1=S2=S3=0;

```
Reader:  loop {S1,S2} : down (S3);
          read;
          {-} : up(S3);
        endloop;
```

```
Writer:  loop {-} : down(S1);
          { S2,S3} : down(S2);
          write;
          { -} : up(S2);
          { -} : up(S1);
        endloop;
```

Anmerkung : {-} bedeutet hier eine leere Vorbedingung, keine Semaphore werden geprüft.

Hier werden "read" und "write" jeweils zu kritischen Abschnitten geklammert mit Operationen, die man naheliegenderweise als read-request, read-end, write-request und write-end bezeichnen könnte.

Dabei fällt auf, daß write-request in zwei Operationen aufgeteilt ist, deren erste keine Vorbedingung hat. Darin drückt sich aus, daß ein Schreiber jederzeit zumindest die Möglichkeit hat, seinen Schreibwunsch anzumelden. Dies mit einer unteilbaren Aktion "write-request" der Form

when B do A ;

darzustellen, wäre unmöglich, da bei Nichterfülltsein der Vorbedingung bbb keine Möglichkeit besteht, den Schreibwunsch eines Schreibers in einer Veränderung der Semaphore darzustellen.

Das Problem könnte man als "Unschärfe des Anfangszeitpunkts" bezeichnen; die Frage ist nämlich: ab welchem genauen Zeitpunkt gilt eine Operation als begonnen, ein kritischer Abschnitt als betreten ?

Diese Unschärfe sieht man auch gut an einem anderen Aspekt des obigen Beispiels:

Folgende Aktionsfolge ist zulässig:

```
(Reader:  {S1,S2} :down(S3) ); (=read-request)
(Writer:   {-} :down(S1)      ); (=write-request,Teil 1)
(Reader:   read                );
```

Obwohl der Schreiber bereits angemeldet ist, kann hier noch ein Leser aktiv werden - denn er hatte sich schon vorher angemeldet. Man muß also das Lesen schon als begonnen betrachten, sobald read-request durchlaufen wurde.

Auf eine explizite Anmeldeoperation kann man so lange verzichten, wie nur Probleme des gegenseitigen Ausschlusses zur Debatte stehen: dann läßt sich mit einer Vorbedingung für die Ausführbarkeit der Aktion "kritischen Abschnitt betreten" alles wesentliche ausdrücken.

Wenn jedoch, wie in obigem Beispiel, Prioritätsgesichtspunkte ins Spiel kommen, dann stellt man Forderungen wie diese:

"Falls eine Operation a1 zur Ausführung ansteht, dann darf keine Operation a2 ausgeführt werden".

Folglich muß das "zur Ausführung anstehen" bereits eine Wirkung auf die Synchronisation haben. Das ist bei dem Konzept von Lipton nur auf dem Umweg einer Zweiteilung der Operation zu erreichen.

In Abschnitt 4.4. werden wir eine Lösung des 2. Leser/Schreiber-Problems mit Fortsetzungsbedingungen angeben.

2.4. Implementierung von P/V-multiple

Dieser Abschnitt diskutiert eine fiktive Implementierung der Operationen P/V-multiple; diese orientiert sich zwar an keiner existierenden Vorlage, ist aber "straight-forward", und zweifellos sieht jeder Implementierungsversuch im wesentlichen so aus; insbesondere gilt das für eine Implementierung nach bekannten algorithmischen Verfahren wie in (MACK83).

Zunächst sei an die abstrakte Definition dieser Operationen erinnert:

```
P(s1,s2, ... ,sn) =
    when (s1 > 0 and ... and sn > 0)
    do   s1:=s1-1, ... , sn:=sn-1 ;

V(s1,s2, ... ,sn) =
    do   s1:=s1+1, ... , sn:= sn+1;
```

In der Implementierung werden als Parameter statt der Semaphore nur deren Nummern genannt.

module PVMULT

```
(*----- lokale Daten -----*)
(* Warteschlange *)
var Q: queue of record A : set of (1..n); (* angeforderte Semaphore *)
      pn: processid (* Prozeßnummer *)
      endrecord;
var s: array (1..n) of integer; (* Semaphore-feld *)

(*----- lokale Funktion, Testen der Semaphore -----*)
function okay(A: set of (1..n)) returns boolean;
  for i in A repeat
    if s(i) = 0 then return false
  end if
end repeat;
return true;
end okay;

(*----- Operationen -----*)
procedure PMULT(A: set of (1..n));
  if okay (A)
  then
    for i in A repeat
      s(i) := s(i) -1;
    endrepeat
```

```

    else
        enqueue (Q, (A, ownpid));
        blockiere (ownpid);
    end if;
end PMULT;

procedure VMULT(A: set of (1..n));
    var done, found :boolean;
    var x: record A: set of (1..n); pn processid endrecord;

    (* Semaphore erhöhen *)
    for i in A repeat
        s(i) := s(i) + 1;
    endrepeat;

    done := false;
    while not done repeat
        (* prüfen, ob Prozesse zu deblockieren sind: die Prozedur
        choose soll dazu aus der Warteschlange Q einen Eintrag x so
        auswählen, daß okay(x.A) gilt und in der Variable found
        mitteilen, ob es einen solchen gibt
        *)
        choose (x, found);
        if found
            then
                remove(Q,x);
                deblockiere (x.pn);
                for i in x.A repeat
                    s(i) := s(i) -1
                endrepeat
            else done := true
            end if;
        endrepeat;
    end VMULT;
end module;

```

Anmerkung:

Das choose-Konstrukt soll hier offenlassen, welcher von ggf. mehreren möglichen Prozessen deblockiert wird. An dieser Stelle versucht man gewöhnlich, die Forderung nach "Fairness" unterzubringen, d.h. die Realisierung von "choose" soll gewährleisten, daß kein Prozeß ausgehungert wird.

Betrachte nun folgendes Prozeßsystem:

(n=2, Anfangswerte $s(1)=s(2)=0$)

```

process P1 : PMULT(1);
process P2 : PMULT(1,2);
process P3 : PMULT(2);
process P4 : VMULT(1); VMULT(2);

```

Hier ist folgender Ablauf möglich:

Aktion	Bemerkung	Werte von s(1), s(2)	Inhalt der Warteschlange: (A ; pn)
(P1:PMULT(1))	P1 blockiert	0,0	(1; 1)
(P2:PMULT(1,2))	P2 blockiert	0,0	(1; 1), (1,2; 2)
(P3:PMULT(2))	P3 blockiert	0,0	(1; 1), (1,2; 2), (2; 3)
(P4:VMULT(1))	zunächst:	1,0	(1; 1), (1,2; 2), (2; 3)
	jetzt wird P1 deblockiert:	0,0	(1,2; 2), (2; 3)
(P4:VMULT(2))	zunächst:	0,1	(1,2; 2), (2; 3)
	jetzt wird P3 deblockiert:	0,0	(1,2; 2)

Hier konkurrieren die beiden Prozesse P1 und P3 mit dem Prozeß P2 um die Semaphore s1 und s2. Wenn P4 diese nacheinander erhöht, kommt es zwangsläufig zu dem geschilderten Ablauf; stattdessen P2 fortzusetzen, ist nicht möglich, obwohl die abstrakte Vorstellung von P/V-multiple dies zuläßt.

Der kritische Punkt ist dabei:

Da die Operation VMULT(1) einen Eintrag in der Warteschlange findet, der deblockiert werden kann, geschieht dies auch! Diese Implementierung berücksichtigt also nicht die Möglichkeit, zunächst keinen der wartenden Prozesse zu deblockieren, um nämlich erst nach der nächsten Freigabe VMULT(2) dann P2 zu deblockieren.

Die obige Behauptung, jeder Implementierungsversuch sehe "im wesentlichen" so aus, bezieht sich auf diese Eigenschaft: wenn man auch versucht, möglichst wenig Strategie-Entscheidung in der Implementierung der Synchronisationsoperationen vorwegzunehmen, wird man doch, sobald es möglich ist, irgendeinen Prozeß deblockieren.

Auch die Annahme, die choose-Funktion sei indeterministisch, löst dieses Problem nicht.

Ein anderes Vorgehen ist kaum sinnvoll, da sonst die Gefahr einer Verklemmung besteht: wie sollte man erreichen, daß ein Prozeß, der zunächst nicht deblockiert wurde, obwohl seine Vorbedingung erfüllt war, doch noch irgendwann zum Zuge kommt ?

Der einzige Ausweg wäre die Einführung einer Zeitbedingung folgender Art:

Nach dem Erhöhen eines Semaphors wartet man (zumindest wenn nicht-erfüllbare Anforderungen vorliegen) eine gewisse Zeit, um weitere V-Operationen zu ermöglichen und im Beispiel dem Prozeß P2 eine Chance zu geben. Trifft aber keine weitere V-Operation (im Beispiel VMULT(2)) ein, so wird ein wartender Prozeß (im Beispiel P1) deblockiert.

Dieser Ansatz ist meines Wissens bisher nicht vorgeschlagen worden. Er hat aus theoretischer Sicht den großen Nachteil, daß die Beschreibung eines Prozeßsystems jetzt von dem zeitlichen Verhalten der Operationen, also auch von solchen Imponderabilien wie der Ausführungsgeschwindigkeit abhängt - die Abstraktion von solchen Äußerlichkeiten ist gerade eine der ursprünglichen Motivationen für die Theorie.

Auch wir wollen diesen zeit-orientierten Ansatz nicht weiter verfolgen, er sollte lediglich die Behauptung erhärten, das obige Beispiel sei typisch für alle bisher untersuchten Implementierungen von P/V-multiple.

Die Kritik an diesen Implementierungen läßt sich in den Satz fassen:

Sie sind in Fällen wie dem des Beispiels zwar sicher (die aktiven Abläufe sind auch im abstrakten P/V-multiple-System aktiv), aber nicht vollständig - es gibt einen Ablauf, der in P/V-multiple aktiv ist, unter der Implementierung aber nicht.

Allerdings ist diese Behauptung formal noch nicht beweisbar, solange kein Modell die Betrachtung der Aufrufzeitpunkte erlaubt. Wenn man davon absieht, wann die Operationen aufgerufen werden, muß man nämlich feststellen, daß der Ablauf

$(P4:VMULT(1)), (P4:VMULT(2)), (P2:PMULT(1,2)),$

der ja durchaus aktiv ist (und zwar ohne Blockierungen), den Ablauf

$V(s1), V(s2), P(s1,s2)$

realisiert.

(Der weiter oben ausführlich betrachtete Ablauf realisiert in diesem Sinn: $V(s1), P(s1), V(s2),$

d.h. einen Aktion zählt dabei erst bei ihrer Deblockierung als ausgeführt.)

Anmerkungen:

1. Die beobachtete Unvollständigkeit gilt nicht nur für Implementierungen von P/V-multiple, sondern natürlich für alle höheren Mechanismen, wie etwa Petri-Netze, aber auch für einige mit P/V-multiple nicht vergleichbare Mechanismen, wie etwa P/V-chunk. Sie tritt immer dann auf, wenn die Ausführungsbedingung einer Operation echt enthalten ist in der Ausführungsbedingung einer anderen Operation.

2. Wie schon in Abschnitt 2.2. gilt auch für diese Implementierung: sie hat eigentlich den Charakter eines F-Systems - die Blockierung erfolgt jeweils am Ende der Prozedur PMULT, während der Aufruf in jedem Fall eine Wirkung hat.

3. Wenn man obige Prozeßsystem geringfügig abändert und annimmt, jeder Prozeß laufe in einer Schleife, dann ist es offensichtlich, daß P2 verhungert.

2.5. Synchronisation und Scheduler

Dieser Abschnitt soll die Begriffe Synchronisation und Scheduling voneinander abgrenzen und ihren Zusammenhang untersuchen.

Ausgangspunkt ist die Vorstellung, daß eine bestimmte Anzahl von Prozessen "quasiparallel" ausgeführt werden, d.h. ihre einzelnen atomaren Aktionen folgen zeitlich durchmischt aufeinander. Dies trifft recht gut auf die Wirklichkeit eines Einprozessor-Systems zu. Auf den Mehrprozessor-Fall soll erst später eingegangen werden (Abschnitt 7.). Jeder einzelne Prozeß legt also die Reihenfolge seiner Aktionen fest in einem sequentiellen Programm, das auch globale Daten bearbeiten darf. Diese prozeßspezifischen Aktionsfolgen werden nun "beliebig" durchmischt (verschränkt, interleaved, merged). Dieses "beliebig" bedeutet zunächst, daß die resultierende Gesamt-Aktionsfolge nicht von den einzelnen Prozessen festgelegt wird; sie ist i.a. nicht vollständig determiniert.

Der Begriff Synchronisation oder Koordinierung asynchroner Systeme bedeutet nun in seinem allgemeinsten Sinne: die Menge der zulässigen Aktionsfolgen wird eingeschränkt, d.h. bestimmte Reihenfolgen werden vorgeschrieben oder verboten.

Schedule

Ein Schedule oder Ablaufplan ist eine Vorschrift zur zeitlichen Abfolge der Bearbeitung mehrerer bereitstehender Prozesse.

Diese Vorschrift wird auch Scheduling-Strategie, diejenige Komponente eines Betriebssystems, welches sie realisiert Scheduler, Prozeßumschalter oder dispatcher (WETT78) genannt. Wir unterscheiden im weiteren nicht streng zwischen diesen verwandten Begriffen.

Ein Scheduler in einem Einprozessor-System teilt den Prozessor den einzelnen Prozessen zu und realisiert so die zeitliche Verschränkung der prozeßspezifischen Aktionsfolgen. Eine atomare Aktion ist i.a. ein Maschinenbefehl, oder auch eine kompliziertere Funktion wie z.B. ein Supervisor-Call (SVC).

Im oben genannten Sinn muß der Scheduler also zu den Synchronisationsmaßnahmen gerechnet werden. Von den eigentlichen Synchronisationsmechanismen unterscheidet er sich aber grundsätzlich dadurch, daß die Anwenderprozesse keinen direkten Einfluß darauf haben - er ist im Betriebssystem selbst angesiedelt. Man kann auch sagen, er übernimmt denjenigen Anteil der Prozeßkoordinierung, der durch die gemeinsame Benutzung (sharing) des Betriebsmittels "Prozessor" nötig wird; die eigentlichen Synchronisationsmechanismen behandeln den Teil, der sich - unabhängig von der Anzahl der Prozessoren - aus inhärenten Eigenschaften des Prozeßsystems selbst ergibt.

Im Sinne von Portabilität, aber auch von begrifflicher Klarheit, ist es unbedingt anzustreben, ein Prozeßsystem unabhängig von der Scheduling-Strategie des Betriebssystems zu formulieren - etwa in einer höheren Programmiersprache, deren Semantik aber dann ohne Rückgriff auf das Betriebssystem definiert sein muß. Mag das auch selbstverständlich erscheinen, für die mit Parallelität zusammenhängenden Konstrukte ist es keineswegs die Regel; selbst die moderne Sprache ADA kennt als "pragma" die Möglichkeit, die "priority" eines Prozesses zu spezifizieren, wobei die genaue Bedeutung implementierungsabhängig ist (PYLE82).

Noch weniger ist diese Betriebssystem-Unabhängigkeit in der Praxis der Prozeß-Datenverarbeitung gegeben: Unterbrechungssperren und Prioritäten sind einfache Beispiele von Koordinierungsmaßnahmen, die dort unentbehrlich erscheinen, wiewohl ihre Wirkung nur jeweils mit Bezug auf ein gegebenes Betriebssystem exakt definiert ist.

Unter Synchronisationsmechanismen im engeren Sinne verstehen wir also solche Konstrukte, die betriebssystem-unabhängig definiert sind oder definiert werden können (dieses Kriterium ist allerdings nicht objektiv verifizierbar, denn prinzipiell kann man jeden maschinennahen Mechanismus allgemein beschreiben, indem man nämlich Betriebssystemteile beschreibt).

Betrachtet man nun vorhandene Implementierungen solcher Mechanismen, so stellt sich heraus: Die Betriebssystemteile zur Synchronisation benutzen die tiefere Ebene des Schedulers. Und zwar verwaltet das Betriebs-

system i.a. Prozeßzustände wie bereit/aktiv/blockiert; der Synchronisationsmechanismus ändert diese Zustände - der Scheduler benutzt sie als Input in der naheliegenden Weise: blockierte Prozesse werden nicht ausgeführt. Die Auswahl zwischen bereiten Prozessen ist abhängig von der jeweiligen Schedule-Strategie.

Simulation

Um zwei Prozeßsysteme P,Q vergleichen zu können, ist der zentrale Begriff "Q simuliert P", d.h. zeigt in gewissem Sinn das gleiche Verhalten. Und zwar soll mit diesem Begriff nicht das Prozeßverhalten erfaßt werden, das unter bestimmten Umgebungsbedingungen, abhängig vom Betriebssystem zustandekommt, sondern nur was in den Prozessen selbst festgelegt ist. Idealerweise sollte also ein Simulationsbegriff aussagen:

Q simuliert P g.d.w.

"Q hat unter jedem Schedule das gleiche Verhalten wie P".

"Gleiches Verhalten" heißt dabei: die Aktionenfolgen entsprechen sich mittels einer Repräsentationsfunktion.

Der Lipton'sche Simulationsbegriff ist genau in diesem Sinn angelegt: es wird insbesondere "Blockadefreiheit" gefordert, d.h. ein Prozeß Qi des simulierenden Systems Q darf sich nicht an einer Stelle blockieren, wo dies der entsprechende Prozeß im simulierten System nicht ebenfalls tun würde - eben weil sonst ein Scheduler auf die beiden Fälle unterschiedlich reagieren könnte, z.B. weil ein Page-Mechanismus den blockierten Prozeß auslagert.

3. Entwicklung und Definition des Modells

Dieser Abschnitt bringt die formalen Definitionen für Prozeßsysteme, Schedules und den darauf aufbauenden Simulationsbegriff. Schließlich werden zwei grundlegende Sätze über diesen Simulationsbegriff bewiesen, Fundamental- und Invarianztheorem, deren Entsprechungen aus (LIPT74) bereits bekannt sind, und die für weitere Anwendungen benötigt werden.

3.1. Prozeßsysteme

Die folgenden Begriffe sind angelehnt an die aus (LIPT74) bekannten, unterscheiden sich aber von diesen in folgendem Punkt:

Die Prozeßzustände "blockiert"/ "nicht blockiert" werden bei uns explizit in das Modell eingeführt, sie gehören per definitionem zum Zustand des Prozeßsystems; bei Lipton kommen diese Zustände zwar vor, aber nur als abgeleitete Begriffe aufgrund der Menge der aktiven Aktionenfolgen.

Die Bedeutung dieser expliziten Betrachtungsweise liegt darin, daß hier ein Prozeß erst dann als blockiert gelten soll, wenn er die blockierende Aktion wenigstens "angefordert" oder "aufgerufen" hat, und nicht bereits, wenn als nächste Aktion eine nicht ausführbare ansteht.

Definition:

Ein Prozeßsystem ist ein Tripel (A, D, w) , wobei gilt:

A ist die Menge der Aktionen, D die Zustandsmenge, w der Anfangszustand. Die Zustandsmenge hat die Form

$$D = D_1 \times D_2 \times \dots \times D_n \times \text{Var} \times \text{Pow}(n),$$

wobei D_i , Adreßraum des Prozesses i , eine endliche Menge ist,

Var der Wertevorrat der Variablen,

und $\text{Pow}(n)$ die Potenzmenge der Menge $\{1, 2, \dots, n\}$;

der Anteil $\text{Pow}(n)$ wird interpretiert als die Menge der nicht blockierten Prozesse.

Eine Aktion $f \in A$ ist wiederum durch folgende sechs Größen gekennzeichnet:

$bz(f) \in \{1, 2, \dots, n\}$, die Prozeß- oder Befehlszählernummer,
 $adr(f) \in Di$, die Adresse im prozeßeigenen Adreßraum,
 wobei $i = bz(f)$,

sowie vier Abbildungen:

$vor(f) \in Abb(Var, \{0, 1\})$,
 die Vorbedingung von f ist ein Prädikat über Var ,

$s(f) \in Abb(Var, Di)$, ($i = bz(f)$),
 die Folgeadresse in Abhängigkeit von den Variablenwerten,

$t(f) \in Abb(Var, Var)$,
 die Wirkung der Aktion auf die Variablen,

und

$u(f) \in Abb(Var \times Pow(n), Pow(n))$,
 die Wirkung auf die Blockierungen.

Die Wirkung einer Aktion f auf einen Zustand

$$z = (L1, L2, \dots, Ln, G, NB) \in D$$

ist dabei wie folgt definiert:

1. f heißt ausführbar im Zustand z , wenn gilt:

$$L(bz(f)) = adr(f) \text{ und}$$

$$bz(f) \in NB \text{ und}$$

$$vor(f)(G) = \text{true},$$

d.h. wenn der Befehlszähler auf f zeigt, der Prozeß nicht blockiert ist, und die Vorbedingung erfüllt ist.

2. Der Folgezustand $f(z) =$

$$(L'1, L'2, \dots, L'n, t(f)(G), u(f)(G, NB)),$$

$$\text{wobei } L'i = s(f)(G) \text{ für } i = bz(f),$$

$$L'i = Li \text{ sonst.}$$

In einem Prozeß darf zu einem Zeitpunkt höchstens eine Aktion ausführbar sein:

es muß gelten

$((f, g \in A) \text{ and}$

$\text{adr}(f) = \text{adr}(g) \text{ and } \text{bz}(f) = \text{bz}(g) \text{ and } f \neq g)$

$\Rightarrow$

$(\text{vor}(f) \text{ and } \text{vor}(g) = \text{false}).$

Wenn hier bewußt zugelassen wird, daß zwei Aktionen mit gleicher Adresse verschiedene Vorbedingungen haben, so geschieht das nur, um später den Aufruf einer Aktion, die sich an einer Vorbedingung blockiert, als eigene Aktion darstellen zu können.

Definition:

Sei $P = (A, D, w)$ ein Prozeßsystem.

Sei $\text{Star}(A)$ die Menge der endlichen Folgen über A . (Die Bezeichnung soll an den üblichen hochgestellten Stern erinnern).

Bezeichne λ die leere Folge.

Die Funktion

$\text{value} \in \text{Abb}(\text{Star}(A), D)$

wird rekursiv wie folgt definiert:

$\text{value}(\lambda) := w;$

ist $\alpha \in \text{Star}(A)$ und $f \in A$, dann sei

$\text{value}(\alpha f) := f(\text{value}(\alpha)).$

Ein aktiver Ablauf in P ist eine Folge

$\alpha = a_1 a_2 \dots a_i \dots a_k$

über A , so daß für jede Aktion a_i gilt:

a_i ist ausführbar im Zustand $\text{value}(a_1 a_2 \dots a_{i-1})$

$(i = 1, 2, \dots, k).$

Die Menge der aktiven Abläufe wird mit $\text{act}(P)$ bezeichnet.

Ein Prozeß i ist blockiert im Zustand $z = (L_1, \dots, L_n, G, NB)$, wenn $i \notin NB$.

Bemerkungen:

In diesem allgemeinen Modell wurden sowohl Vorbedingungen als auch explizite Blockaden vorgesehen, um einen allgemeinen Rahmen auch zum Vergleich von V- und F-Systemen zu schaffen.

Nach dieser Definition kann also ein Prozeß direkt Einfluß nehmen auf die Blockierungen sämtlicher Prozesse, und es ist erlaubt, daß in einem Prozeß keine Aktion ausführbar ist, obwohl er nicht blockiert ist. Diese unbequemen Eigenschaften werden in den spezielleren Systemen, die in 3.2. definiert sind, eingeschränkt.

Die obige allgemeine Definition legt vor allem diejenigen Eigenschaften eines Prozeßsystems fest, die als "Schnittstelle" zum Scheduler benötigt werden, nämlich die Blockierungen in Abhängigkeit vom Ablauf. Dies wird im Folgenden präzisiert:

Definition:

Sei $\alpha \in \text{act}(P)$. Die zu α gehörige Schaltfolge
 $\text{sf}(\alpha)$

ist ein Element von $\text{Star}(\{1, \dots, n\} \times \text{Pow}(n))$, definiert wie folgt:

$\text{sf}(\lambda) := \lambda$;

$\text{sf}(\alpha f) := \text{sf}(\alpha) (bz(f), NB(\text{value}(\alpha f)))$,

also eine Folge von Paaren aus Prozeßnummern und Prozeßmengen, die angibt, welche Prozesse aktiv und welche nicht blockiert sind.

3.2. F- und V-Systeme

Es werden zwei spezielle Klassen von Prozeßsystemen definiert.

Definition:

Ein F-System ist ein Prozeßsystem $P = (A, D, w)$ mit folgenden Eigenschaften:

1. Die Variablenmenge Var hat die Form

$$Var = Va \times Pow(Va)^n ;$$

dabei steht Va für den Wertevorrat der eigentlichen Variablen,
 $Pow(Va)^n$ für das n -fache cartesische Produkt der
 Potenzmenge von Va ;

Die i -te Komponente wird interpretiert als Fortsetzungsbedingung
 des i -ten Prozesses - eine Teilmenge von Va kann ja als Prädikat
 über Va aufgefaßt werden.

Teilmengen werden hier mit Prädikaten identifiziert.

2. für jedes $f \in A$ ist $vor(f) = true$,

d.h. es gibt keine nichttrivialen Vorbedingungen.

3. für jedes $f \in A$ und jedes

$$(G, (Pr1, Pr2, \dots, Prn)) \in Var,$$

$$(G, (Pr1', Pr2', \dots, Prn')) \in Var,$$

ist

$$s(G, (Pr1, Pr2, \dots, Prn)) = s(G, (Pr1', Pr2', \dots, Prn')),$$

d.h. die Adreßweilerschaltung ist nur von den eigentlichen
 Variablen abhängig.

4. für jedes $f \in A$ hat die Funktion $t(f)$ die Form

$$t(G, (Pr1, Pr2, \dots, Prn)) = (t1(G), t2(G, (Pr1, \dots, Prn))),$$

wobei $t1(f) \in Abb(Va, Va)$,

$$t2(f) \in Abb(Var, Pow(n)^n),$$

und $t2$ wie folgt definiert ist:

$$t2(G, (Pr1, \dots, Prn)) := (Pr1', Pr2', \dots, Prn'),$$

$$Pr1' = \text{fort}(f) \text{ für } i = bz(f),$$

$$Pr1' = Pr1 \text{ sonst.}$$

Das Prädikat $\text{fort}(f) \in Pow(Va)$ heißt dabei Fortsetzungs-
bedingung von f .

D.h. die eigentlichen Variablen dürfen in beliebiger Weise durch die Funktion t_1 manipuliert werden, die Fortsetzungsbedingungen nur in kontrollierter Weise: die Funktion t_2 drückt aus, daß ein Prozeß seine Fortsetzungsbedingung eintragen, die anderen aber nicht manipulieren darf.

5. für jedes $f \in A$ hat die Funktion $u(f)$ die Form

$$u(G, (Pr_1, \dots, Pr_n), NB) = NB',$$

wobei $i \in NB'$ g.d.w.

$$(t_1(G) \in Pr_i' \text{ or } i \neq bz(f) \text{ and } i \in NB);$$

Ein Prozeß kann also einen anderen wohl deblockieren, aber nicht blockieren. Deblockiert wird jeder Prozeß, dessen Fortsetzungsbedingung erfüllt ist; blockiert wird der aktive Prozeß, falls die seinige nicht erfüllt ist.

6. Für jede Aktion $f \in A$ gilt

$$\text{entweder } \text{fort}(f) = Va,$$

dann heißt f nicht-synchronisierend und wird so geschrieben:

$$\underline{\text{do}} G := t_1(G) \underline{\text{goto}} s(G);$$

oder $\text{fort}(f)$ ist eine echte Teilmenge von Va ,

$$\text{dann ist } s(f) = \text{adr}(f)+1,$$

f heißt synchronisierend und wird so geschrieben:

$$\underline{\text{do}} G := t_1(G) \underline{\text{wait}} \text{fort};$$

Diese Einschränkung besagt, daß synchronisierende Aktionen nicht zugleich verzweigend sein können; sie dient nur der Übersichtlichkeit.

Definition:

Ein V-System ist ein Prozeßsystem $P = (A, D, w)$ mit folgenden Eigenschaften:

1. Die Variablenmenge Var hat die Form

$$\text{Var} = Va \times \text{Pow}(Va)^n;$$

dabei steht Va wie oben für den Wertevorrat der eigentlichen Variablen,

$\text{Pow}(Va)^n$ für das n -fache cartesische Produkt der Potenzmenge von Va ;

Die i -te Komponente wird interpretiert als Vorbedingung der nächsten Aktion des i -ten Prozesses.

2. Für jedes $i = 1, 2, \dots, n$ und jeden Befehlszählerstand $L_i \in D_i$ gibt es genau zwei Aktionen $f \in A$ mit
 $bz(f) = i$ and $adr(f) = L_i$.
 Diese beiden werden f -call und f -exec genannt und durch folgende Forderungen festgelegt.
 3. $\text{vor}(f\text{-call}) = \text{not vor}(f\text{-exec})$,
 d.h. die Vorbedingungen sind komplementär.
 4. $s(f\text{-call}) = \text{adr}(f)$,
 d.h. f -call ändert nicht den Befehlszählerstand.
 Daß dennoch keine Endlosschleife zustandekommt, liegt an der unten zu definierenden Blockierung.
 - 5.a) $t(f\text{-call})$ hat folgende Form:

$$t(G, (Pr_1, Pr_2, \dots, Pr_n)) = (G, (Pr_1', Pr_2', \dots, Pr_n'))$$
 wobei $Pr_j' = \text{vor}(f\text{-exec})$ für $j=i$,
 $Pr_j' = Pr_j$ sonst.
 - 5.b) $t(f\text{-exec})$ hat folgende Form:

$$t(G, (Pr_1, Pr_2, \dots, Pr_n)) = (t_1(G), (Pr_1', Pr_2', \dots, Pr_n'))$$
 wobei $Pr_j' = \text{true}$ für $j=i$,
 $Pr_j' = Pr_j$ sonst.
- Nur f -exec darf also die eigentlichen Variablen verändern, und jeder Prozeß darf nur seine eigene Vorbedingung manipulieren.
- 6.a) $u(f\text{-call})$ hat folgende Form:

$$u(G, (Pr_1, Pr_2, \dots, Pr_n), NB) = NB - \{i\},$$
 d.h. der Prozeß i wird blockiert.

6.b) $u(f\text{-exec})$ hat folgende Form

$$u(G, (Pr_1, Pr_2, \dots, Pr_n), NB) = NB',$$

wobei

$$j \in NB' \Leftrightarrow (j = i \text{ or } t1(G) \in Pr_j),$$

d.h. alle Blockaden werden aufgrund der Prädikate Pr_j neu berechnet.

Die scheinbar umständliche Einführung der call-Aktionen $f\text{-call}$ bewirkt nun, daß in einem nicht explizit blockierten Prozeß jederzeit (genau) eine Aktion ausführbar ist, und zwar:

- entweder eine Aktion $f\text{-call}$, falls die Vorbedingung von $f\text{-exec}$ nicht erfüllt ist - dabei blockiert sich der Prozeß - ,

- oder eine Aktion $f\text{-exec}$, falls die Vorbedingung erfüllt ist.

Keineswegs muß vor $f\text{-exec}$ immer $f\text{-call}$ ausgeführt werden; falls die Vorbedingung gilt, wenn der Prozeß aktiv wird, wird gleich $f\text{-exec}$ ausgeführt.

Die Eigenschaft

"in einem nicht blockierten Prozeß ist genau eine Aktion ausführbar"

ist notwendig wegen der Vorstellung vom Schedule:

Dieser kann jederzeit einen Prozeß, der nicht blockiert ist, aktivieren. Er kann aber nicht dessen nächste Aktion kennen, also nicht wissen, ob der Prozeß sich blockieren wird.

Die Aktion $f\text{-call}$ stellt also den Fall dar, daß einem Prozeß vom Scheduler der Prozessor zugeteilt wird, der Prozeß sich aber wegen nicht erfüllter Vorbedingung blockiert.

3.3 Schedules

Um ein auf Schedules begründetes Vergleichskonzept für Synchronisationsmechanismen zu entwickeln, wird zunächst der Begriff "Schedule" formal definiert. Außerdem benötigen wir "Klassen" von Schedules.

Wie bereits erwähnt, besteht die Schnittstelle zwischen Prozeßsystem und Scheduler aus den sogenannten Schaltfolgen:

Definition:

Der Raum der Schaltfolgen für n Prozesse ist

$$SF_n = \text{Star} (\{1, \dots, n\} \times \text{Pow}(n));$$

Sei

$$(p_1, NB_1)(p_2, NB_2) \dots (p_k, NB_k)$$

eine Schaltfolge. Diese heißt zulässig, wenn für $i = 1, 2, \dots, k-1$ gilt:

$$p(i+1) \in NB_i;$$

da NB als Menge nicht blockierter Prozesse, p_i als Nummer des aktivierten Prozesses interpretiert wird, heißt dies: es dürfen nur nicht blockierte Prozesse aktiviert werden.

Die Menge der zulässigen Schaltfolgen in SF_n heißt ZSF_n .

Definition:

Ein Schedule für n Prozesse ist eine Teilmenge S von ZSF_n mit folgenden Eigenschaften:

(Verklemmungsfreiheit:)

$$\text{Ist } (p_1, NB_1)(p_2, NB_2) \dots (p_k, NB_k) \in S,$$

$$\text{und } NB_k \neq \emptyset,$$

$$\text{so gibt es ein } p \in \{1, \dots, n\}, \text{ und ein } NB \in \text{Pow}(n),$$

$$\text{so daß}$$

$$(p_1, NB_1)(p_2, NB_2) \dots (p_k, NB_k)(p, NB) \in S.$$

(Unabhängigkeit:)

$$\text{Ist } (p_1, NB_1)(p_2, NB_2) \dots (p_k, NB_k) \in S,$$

$$\text{und } NB \in \text{Pow}(n),$$

$$\text{so ist } (p_1, NB_1)(p_2, NB_2) \dots (p_k, NB_k) \in S.$$

(Anfangsstück-Erhaltung:)

Sind $\alpha, \beta \in SF_n$, und

$\alpha \beta \in S$,

so ist auch $\alpha \in S$.

Die drei Eigenschaften besagen,

- daß der Scheduler immer einen Prozeß aktiviert, falls nicht das Prozeßsystem selbst verklemmt ist,
- daß der Scheduler keinen Einfluß auf die Blockierungen hat,
- und daß jedes Anfangsstück einer möglichen Schaltfolge selbst möglich ist.

Die Elemente von S heißen mögliche Schaltfolgen in S .

Es dürfte bereits klar sein, wie ein Scheduler für n Prozesse mit einem Prozeßsystem aus n Prozessen zusammenwirkt:

Definition:

Sei P ein Prozeßsystem aus n Prozessen, S ein Scheduler für n Prozesse, α ein aktiver Ablauf in P . α heißt möglicher Ablauf von P unter S , falls

$sf(\alpha) \in S$,

wobei sf die in 3.1. definierte Funktion ist.

Zunächst bleibt festzuhalten: aufgrund unserer Definition von Schaltfolge unterscheidet der Scheduler nicht zwischen verschiedenen Aktionen eines Prozesses, sondern nur zwischen den einzelnen Prozessen. Er hat auch keinen Zugang zu den Daten des Prozeßsystems, sondern operiert ausschließlich auf Schaltfolgen. Dies stellt gewiß eine einigermaßen realistische Abstraktion dar. Es wäre zwar denkbar, weitere Information für das Scheduling heranzuziehen, etwa E/A-Aktivitäten von CPU-Aktivitäten eines Prozesses zu unterscheiden, aber auf jeden Fall gibt es eine ähnliche Abstraktion.

Dagegen ist es fraglich, ob die Schaltfolgen nicht immer noch zu viel Information enthalten:

Das bisherige Konzept läßt ja zu, daß ein Scheduler z. B. die Folge

$(1, \{1, 2\}) (2, \{1, 2\})$

enthält, nicht aber die Folge

$(1, \{1, 2\}) (1, \{1, 2\}) (2, \{1, 2\})$.

Das bedeutet, der Scheduler kann unterscheiden, wie viele Aktionen (Maschinenbefehle ?) ein Prozeß nacheinander ausgeführt hat. Dies wäre angebracht, wenn man von der Vorstellung ausgeht, das Betriebssystem teile den Prozessor nach jedem einzelnen Befehl neu zu - ein durchaus denkbare Verfahren. Tatsächlich geschieht aber gewöhnlich anderes: ein einmal aktivierter Prozeß wird erst unterbrochen und ggf. verdrängt, wenn gewisse Ereignisse eintreten, etwa nach Ablauf einer Zeitscheibe oder wenn ein höherpriorer Prozeß ablauffähig wird. Er hat in dieser Zeit eine unbestimmte Anzahl von Aktionen ausgeführt.

Man könnte sich auch nur für solche Schedules interessieren, die ohne Zeitscheiben arbeiten, die also nur bei Änderung einer Blockade eine neue Prozessorvergabe durchführen.

Um diese und auch noch weitere spezielle Arten von Schedules untersuchen zu können, wird das Konzept der "Scheduleklasse" eingeführt.

Definition:

Eine Scheduleklasse K für n Prozesse ist ein Paar (M, wk) , wobei

M ein Schedule für n Prozesse, und

wk eine transitive und reflexive Relation auf SF_n ist.

Wir schreiben $a \text{ } wk \text{ } b$ (lies "weaker"), wenn a und b in der Relation stehen.

M heißt der größte Schedule der Klasse K . Die Bedeutung von wk ergibt sich aus folgendem:

Ein Schedule S für n Prozesse gehört zur Klasse K , wenn

$$S \subset M$$

und (für $\alpha, \beta, \gamma \in SF_n$):

$(\beta \text{ } wk \text{ } \gamma \text{ and } \alpha \text{ } wk \text{ } \beta)$

$\Rightarrow (\alpha \text{ } wk \text{ } \gamma)$.

Mit anderen Worten, $\alpha \text{ } wk \text{ } \beta$ heißt, ein Ablauf, der nach β möglich ist, ist auch nach α möglich.

Die Relation wk muß nicht antisymmetrisch sein, aber wenn man definiert:

$a \dot{\sim} b$ g.d.w. $(a \text{ wk } b \text{ and } b \text{ wk } a)$,
dann ist $\dot{\sim}$ eine Äquivalenzrelation, und die von wk auf den
Äquivalenzklassen induzierte Relation ist eine partielle Ordnung.

Äquivalenz von Schaltfolgen bedeutet dann, daß ein Schedule (der
betrachteten Klasse) diese nicht unterscheiden kann.

Beispiel:

Die Klasse NZ_n der nicht zählenden Schedules ist definiert durch:

$M = SF_n$;
 $a \text{ wk } b \Leftrightarrow b \text{ wk } a$;
 $(i, NB) \text{ wk } (i, NB)(i, NB)$;
 $\alpha \text{ wk } \alpha$;
 $(\beta_1 \text{ wk } \beta_2) \Rightarrow$
 $((\alpha \beta_1 \gamma) \text{ wk } (\alpha \beta_2 \gamma))$;
 wk ist der transitive Abschluß dieser Axiome.

Hier ist wk selbst eine Äquivalenzrelation und besagt:

Zwei Schaltfolgen werden als gleich betrachtet, wenn sie sich nur durch
Wiederholung einzelner Glieder unterscheiden.

Die Menge M ist hier die größtmögliche, stellt also keine Einschränkung
dar.

Anders im folgenden Beispiel:

Die Klasse ZF der zeitfreien Schedules wird in Abschnitt 6.2.1.
definiert; der zugehörige größte Schedule ist:

$M = \{$
 $(p_1, NB_1)(p_2, NB_2) \dots (p_k, NB_k) \in ZSF_n$;
 für $i=1, 2, \dots, k-2$ gilt:
 $p(i+1) = p(i+2) \text{ or } NB_i \neq NB(i+1)$
 $\};$

D.h. ein Prozeßwechsel ist nur möglich, wenn sich die Menge NB der
nicht blockierten Prozesse geändert hat.

3.4. Der Simulationsbegriff

Für den Vergleich von Prozeßsystemen und von Synchronisationsmechanismen spielt der Simulationsbegriff die zentrale Rolle. Selbst die Frage, ob ein gegebenes Prozeßsystem Q ein gegebenes Synchronisationsproblem "löst", läßt sich unmißverständlich nur dann beantworten, wenn das Problem selbst formal als Prozeßsystem P gegeben ist und man die Frage reduziert zu : Gilt " Q simuliert P " ?.

Dies wurde von Lipton (LIPT74) erkannt; es sei daran erinnert, daß vor jener Arbeit mangels eines strengen Lösungsbegriffs keine Klarheit über die unterschiedliche Mächtigkeit der verschiedenen Synchronisationsmechanismen bestand. Der in der Literatur dokumentierte Gelehrtenstreit zwischen P.B. Hansen auf der einen und P.J. Courtois, F. Heymans, D.L. Parnas auf der anderen Seite (COUR71, HANS72, COUR72, HANS73) beruhte auf dieser Unklarheit (zitiert nach BATH82).

Wie schon früher angedeutet, soll die beabsichtigte Bedeutung unseres Simulationsbegriffes in etwa sein:

" Q zeigt unter jedem Schedule das gleiche Verhalten wie P ."

Genauer: man wird sich nicht immer für "jeden" Schedule interessieren, sondern für eine gewisse Klasse - dazu wurde im letzten Abschnitt die Scheduleklasse definiert.

" Q simuliert P " wird zunächst in bezug auf einen einzelnen Schedule, dann in bezug auf eine Scheduleklasse definiert, schließlich die Beziehung dieser beiden Begriffe untersucht.

Seien also $P = (A1, D1, w1)$, $Q = (A2, D2, w2)$ F-Systeme mit n Prozessen. (Eine Verallgemeinerung auf F- und V-Systeme betrachten wir in 5.1.).

Definition:

Eine Abbildung $r \in \text{Abb}(\text{Star}(A2), \text{Star}(A1))$ heißt Realisierung von P durch Q , wenn r ein Halbgruppen-Homomorphismus und strukturerhaltend ist; strukturerhaltend heißt

für $a, b \in A2$ gilt:

$$bz(r(a)) = bz(r(b)) \Leftrightarrow bz(a) = bz(b),$$

d.h. r bildet Prozesse bijektiv aufeinander ab.

Diese Einschränkung wird weiter unten noch diskutiert.

Eine Aktion $f \in A2$ heißt beobachtbar, wenn $r(f) \neq \lambda$.

Eine Aktion $f \in A2$ heißt Buchführungsaktion, wenn $r(f) = \lambda$.

Definition:

Sei S ein Schedule für n Prozesse. Q simuliert P bezüglich S durch r , wenn r eine Realisierung von P durch Q ist, und wenn gilt:

(1) r ist vollständig bezüglich S , d.h.

$$r \{ \alpha; \alpha \in \text{act}(Q) \text{ and } sf(\alpha) \in S \}$$

$$\supseteq$$

$$\{ \beta; \beta \in \text{act}(P) \text{ and } sf(\beta) \in S \}$$

(2) r ist sicher bezüglich S , d.h.

$$r \{ \alpha; \alpha \in \text{act}(Q) \text{ and } sf(\alpha) \in S \}$$

$$\subseteq$$

$$\{ \beta; \beta \in \text{act}(P) \text{ and } sf(\beta) \in S \}$$

Mit anderen Worten, r bildet die unter S möglichen Abläufe in Q auf die unter S möglichen Abläufe in P ab.

Es wäre nun naheliegend, in der vorstehenden Definition S universell zu quantifizieren, und das Konzept

" Q simuliert P bezüglich aller Schedules der Klasse K "

zum Ausgangspunkt der Untersuchung zu machen.

Leider ist mit diesem Konzept schwierig etwas zu beweisen, daher führen wir den folgenden, geringfügig strengeren Begriff ein.

Definition:

Sei $K = (M, wk)$ eine Scheduleklasse und r eine Realisierung von P durch Q .

Q simuliert P bezüglich der Klasse K durch r , wenn gilt

(1) r ist vollständig bezüglich K , d.h.

für jedes $\beta \in \text{act}(P)$ mit $sf(\beta) \in M$

gibt es ein $\alpha \in \text{act}(Q)$,

so daß $r(\alpha) = \beta$

und $sf(\alpha) \text{ wk } sf(\beta)$.

- (2) r ist sicher bezüglich der Klasse K , d.h.
 für jedes $\alpha \in \text{act}(Q)$ mit $\text{sf}(\alpha) \in M$
 gibt es ein $\beta \in \text{act}(P)$,
 so daß $r(\alpha) = \beta$
 und $\text{sf}(\beta) \text{ wk } \text{sf}(\alpha)$.

Die Relation, die hier für die Schaltfolgen gefordert wird, bedeutet ja: (1) jeder Schedule, der β erlaubt, erlaubt auch α , und (2) umgekehrt.

In direkter Anlehnung an (LIPT74) (vgl. Abschnitt 1.1.) definieren wir noch Promptheit:

Definition:

Eine Realisierung r von P durch Q ist prompt, wenn eine Konstante c existiert, so daß für alle aktiven Abläufe α von Q gilt:

$$1 + \#r(\alpha) > c \times \#\alpha.$$

Falls Q dann P durch r bezüglich einer Klasse K (bzw. eines Schedules S) simuliert, dann sagen wir, Q simuliert P durch r prompt bezüglich K (bzw. S).

Um die Definition der Simulation zu rechtfertigen, soll der folgende Satz den Zusammenhang des klassenbezogenen Simulationsbegriff mit der Vorstellung "Simulation bezüglich jedes Schedules einer Klasse" herstellen.

Seien also P, Q Prozeßsysteme und r eine Realisierung von P durch Q . Sei $K = (M, \text{wk})$ eine Scheduleklasse.

Satz:

Falls Q simuliert P bezüglich K durch r , dann gilt für jeden Schedule S der Klasse K :

Q simuliert P bezüglich S durch r .

Beweis:

Sei S ein Schedule der Klasse K .

Zu zeigen ist: r ist vollständig und sicher bezüglich S .

"vollständig":

Sei $\alpha \in \text{act}(P)$, $\text{sf}(\alpha) \in S$,

dann gibt es wegen der Vollständigkeit der Simulation bezüglich K einen Ablauf $\beta \in \text{act}(Q)$ mit:

$$r(\beta) = \alpha, \quad \text{sf}(\beta) \text{ wk } \text{sf}(\alpha).$$

Nach der Definition von " S gehört zu K " folgt aus

$$\text{sf}(\alpha) \in S \quad \text{und} \quad \text{sf}(\beta) \text{ wk } \text{sf}(\alpha),$$

daß $\text{sf}(\beta) \in S$.

"sicher":

Sei $\beta \in \text{act}(Q)$, $\text{sf}(\beta) \in S$,

dann gilt wegen der Sicherheit der Simulation bezüglich K :

$$r(\beta) = \alpha, \quad \text{sf}(\alpha) \text{ wk } \text{sf}(\beta).$$

Nach der Definition von " S gehört zu K " folgt aus

$$\text{sf}(\beta) \in S \quad \text{und} \quad \text{sf}(\alpha) \text{ wk } \text{sf}(\beta),$$

daß $\text{sf}(\alpha) \in S$.

Q.E.D.

Die Umkehrung dieses Satzes gilt leider nicht ohne Einschränkung, wie das folgende Beispiel zeigt. Betrachte zwei Prozeßsysteme:

P: integer z init 0;

process P1: loop (a1: do $z:=0$ wait $z=1$) endloop;

process P2: loop (a2: do $z:=1$ wait $z=0$) endloop;

Q: integer z init 0;

process Q1: loop (b1: do $z:=0$ wait $z=1$) endloop;

process Q2: loop (b21: do $z:=1$; b22: do wait $z=0$) endloop;

(Hier sind $a_1, a_2, \dots$ Namen der jeweiligen Aktionen.)

Man sieht leicht, daß durch die Realisierung

$$r(b1) = a1; \quad r(b21) = a2; \quad r(b22) = \text{lambda};$$

Q P bezüglich jedes Schedules simuliert, und zwar einfach deshalb, weil der Schedule hier keine Rolle spielt; das Prozeßsystem P ist nahezu deterministisch, d.h. wenn erst eine Aktion ausgeführt wurde, gibt es keine Wahlmöglichkeit mehr.

Dagegen unterscheiden sich P und Q sehr wohl, wenn man die Blockaden betrachtet, insbesondere gibt es in Q Zustände, in denen kein Prozeß blockiert ist, in P gibt es solche - abgesehen vom Anfangszustand - nicht. Dieses unterschiedliche Verhalten wird von dem klassenbezogenen Simulationsbegriff erfaßt:

Sei etwa $K = \mathbb{N}Z$ die Klasse der nicht-zählenden Schedules; betrachte den Ablauf in Q:

$$\alpha = b1 \ b21 \ b22$$

Es ist

$$sf(\alpha) = (1, \{2\}) (2, \{1, 2\}) (2, \{1\}),$$

$$r(\alpha) = a1 \ a2,$$

$$sf(r(\alpha)) = (1, \{2\}) (2, \{1\}).$$

Also nicht $sf(r(\alpha)) \leq_K sf(\alpha)$,

r ist nicht sicher, Q simuliert P nicht bezüglich K.

Vermutung:

Eine Umkehrung des obigen Satzes gilt mit der zusätzlichen Voraussetzung, daß es in P zu jedem erreichbaren Zustand verschiedene Fortsetzungen gibt.

Zur Forderung der Strukturhaltung

Gegenüber Lipton's Definition ist die Forderung der Strukturhaltung gewiß eine Einschränkung. Dort wird nur "Prozeßtreue" gefordert, ein Prozeß darf nicht zur Simulation mehrerer beitragen, aber ein Prozeß kann durch mehrere simuliert werden. In der Praxis kann das auftreten, wenn bestimmte Teile eines Anwenderprozesses (etwa Ein-/Ausgabe) an Betriebssystem-Prozesse delegiert werden.

Dennoch erscheint uns die Beschränkung auf Bijektionen der Prozesse aus folgenden Gründen gerechtfertigt:

a) Wird ein Prozeßsystem in einer höheren Sprache beschrieben und von einem Compiler realisiert, dann erhält die Compilation in aller Regel die Prozeßstruktur. Der Compiler wird keinen Prozeß in mehrere zerlegen.

b) Gerade die Situation von Betriebssystem-Prozessen, die für den Anwender tätig sind, wird auch von der schwächeren Forderung der Prozeßtreue nicht abgedeckt: denn ein solcher Prozeß ist in der Regel für mehrere Anwender tätig, verletzt also diese Forderung. Diese Situation in einem Modell zu erfassen, wäre gewiß eine eigene Untersuchung wert.

c) Unser Ansatz der Einbeziehung von Schedules macht es notwendig, die Prozeßsysteme P und Q unter dem gleichen Schedule betrachten zu können. Es ist aber schwierig, ein allgemeines Konzept von "Schedule" so zu definieren, daß es für unterschiedliche Anzahlen von Prozessen zutrifft. Der naheliegende Weg wäre eine Forderung der Art

"ein Prozeß P_i ist genau dann blockiert, wenn alle ihn simulierenden Prozesse blockiert sind".

Dies führt aber zu Schwierigkeiten im Fall von unbeobachtbaren Aktionen.

3.5. Diskussion verschiedener Scheduleklassen

Die Betrachtung von nicht-zählenden Schedules wurde bereits motiviert. Im Hinblick auf die Simulation läßt sich jetzt feststellen: Ist die Relation wk zu fein oder gar die Identität, dann wird der Simulationsbegriff zu streng.

Das gilt selbst in einem Prozeßsystem ohne synchronisierende Aktionen, also ohne jede Blockierung. Sobald nämlich unbeobachtbare Aktionen vorkommen, braucht das simulierende System mehr Einzelschritte als das simulierte. Ein "zählender" Schedule kann dies unterscheiden und verschieden reagieren.

Das folgende Beispiel soll zeigen, daß selbst nicht-zählende Schedules noch zu streng sein können.

Auch dieses Beispiel kommt ohne synchronisierende Aktionen aus. Es genügt sogar, die Aktionen nur zu benennen, ohne ihre Wirkung anzugeben: Die Prozeßsysteme P und Q unterscheiden sich nur durch eine unbeobachtbare Aktion:

P: process P1: a1; a2;
 process P2: b1; b2;
 process P3: c2;

Q: process Q1: a1'; a2';
 process Q2: b1'; b2';
 process Q3: c1'; c2';

Eine Realisierung von P durch Q ist r :

$r(c1') = \text{lambda}$; $r(a1') = a1$; $r(a2') = a2$ usw.

Definiere nun den Schedule $S = RR$ ("round robin") wie folgt:

a) $(1, NB) \in S$ für alle $NB \in \{1, 2, 3\}$.

b) $(\alpha(i, NB) \in S, NB \neq 0) \Rightarrow$
 $(\alpha(i, NB)(j, NB') \in S, \text{ wo }$
 $j \in NB \text{ and } ,$
 $((i < k < j) \text{ or }$
 $(i > j \text{ and } (k > i \text{ or } k < j))$
 $)$
 $\Rightarrow (k \notin NB)$
 $)$

c) S besteht genau aus den Schaltfolgen, die aus den Axiomen a) und
 b) mittels der Relation $wk(NZ)$ erzeugt werden.

Anschaulich bedeutet das: RR aktiviert jeweils den nächsten nicht
 blockierten Prozeß in zyklischer Reihenfolge; überspringen ist nicht
 möglich, wohl aber Wiederholung des gleichen Prozesses, da es ein
 nicht-zählender Schedule sein soll.

Betrachte nun das Prozeßsystem Q unter dem Schedule RR :
 (da keine Blockaden auftreten, setze $NB = \{1, 2, 3\}$).

Für den Ablauf

$\alpha = a1' b1' c1' a2'$

gilt

$sf(\alpha) = (1, NB) (2, NB) (3, NB) (1, NB) \in RR$

aber

$r(\alpha) = a1 b1 a2,$

$sf(r(\alpha)) = (1, NB) (2, NB) (1, NB) \notin RR.$

Folglich simuliert Q nicht P bezüglich RR .

Man wird daher die Relation wk weiter vergrößern, so daß z. B. gilt:

$(1, NB) (2, NB) (1, NB) \quad wk \quad (1, NB) (2, NB) (3, NB) (1, NB)$

Das hat zur Folge, daß mit dem Ablauf α auch $r(\alpha)$ möglich ist,
 wenn sich dieses nur durch Weglassen unbeobachtbarer Aktionen unter-
 scheidet.

Definition:

Die Scheduleklasse SNZ_n ("schwach nicht-zählend") ist definiert durch

$SNZ = (M, wk)$ mit

$M = ZSF_n$,

wk ist der transitive Abschluß folgender Axiome:

$\alpha wk \alpha$,

$(i, NB) wk (i, NB)(j, NB)$, (Adsorption)

$(i, NB)(i, NB) wk (i, NB)$, (nicht-zählend)

$(\beta_1 wk \beta_2) \Rightarrow ((\alpha \beta_1 \gamma) wk (\alpha \beta_2 \gamma))$

(Verträglichkeit mit Monoid-Struktur).

Im Gegensatz zu NZ_n ist hier die Relation wk keine Äquivalenz mehr; die Adsorption bedeutet: jede Aktion, die keine Blockaden ändert, kann aus der Sicht des Schedulers weggelassen werden - der entstehende Ablauf ist immer noch möglich, falls er es vorher war. Nicht-zählend besagt, wo ein Prozeß überhaupt aktiv sein kann, da darf er auch beliebig viele Aktionen ausführen.

Der Unterschied zu NZ läßt sich auch so charakterisieren: NZ unterscheidet nicht zwischen 1, 2, 3... -maligem Schalten eines Prozesses, wohl aber zwischen 0- und 1-maligem; für SNZ stehen auch "0" und "1" in der Relation, aber nur in einer Richtung :

$0 \cdot 1 = 2 = 3 = \dots$

Hier bewährt sich die Wahl von wk als nur transitiv, anstelle einer Äquivalenz.

Die informale Charakterisierung gilt immer nur für den Fall, daß sich keine Blockierung ändert. Abläufe, die sich in der Folge der NB unterscheiden, stehen nicht in der Relation wk .

Beispiele ($n=3$):

$(1, \{1, 2\}) (2, \{1, 2, 3\}) (2, \{1, 3\})$

wk

$(1, \{1, 2\}) (1, \{1, 2\}) (2, \{1, 2, 3\}) (2, \{1, 3\}) (2, \{1, 3\})$

wk

$(1, \{1, 2\}) (2, \{1, 2, 3\}) (2, \{1, 3\})$;

D.h. diese beiden Folgen sind äquivalent (bereits unter NZ).

$(1, \{1, 2\}) (1, \{1, 2\}) (2, \{1, 2, 3\})$

wk

$(1, \{1, 2\}) (2, \{1, 2\}) (1, \{1, 2\}) (2, \{1, 2, 3\})$

Die Umkehrung gilt nicht, und diese beiden Folgen sind unter NZ nicht vergleichbar.

Welche Rollen spielen nun die Kenngrößen M und wk einer Scheduleklasse für unseren Simulationsbegriff ?

Aus den Definitionen sieht man: Simulation wird um so leichter, je kleiner der größte Schedule M und je größer (größer) die Relation wk gewählt wird. Wenn, wie im Fall $K=SNZ$ oder NZ , $M = ZSF_n$ gewählt wird (also die größtmögliche Wahl), so hat man alle zulässigen Schaltfolgen, also alle aktiven Abläufe der beiden zu vergleichenden Prozeßsysteme zu betrachten. Wird M eingeschränkt, heißt das, Abläufe in P, deren Schaltfolgen nicht zu M gehören, brauchen auch von Q nicht simuliert zu werden (Abschwächung der Vollständigkeit) und Abläufe in Q, deren Schaltfolgen nicht zu M gehören, brauchen in P nicht aktive Folgen zu realisieren (Abschwächung der Sicherheit).

Wird wk klein (groß) gewählt, so stellt man strenge (weniger strenge) Anforderungen an den Zusammenhang der Schaltfolgen alpha und $r(\alpha)$, und zwar in den beiden Richtungen "vollständig" und "sicher".

So gilt z.B. für die eingangs eingeführten Systeme P und Q:

Q simuliert P bezüglich der Klasse SNZ durch r.

Beweis:

"vollständig"

Zu einem beliebigen aktiven Ablauf beta in P betrachte denjenigen Ablauf alpha in Q, der durch Ersetzung von

$c1' c2'$ für $c2$,

$a1'$ für $a1$,

$a2'$ für $a2$ usw.

aus beta entsteht. Dann ist $r(\alpha) = \beta$ und es entsteht $sf(\alpha)$ aus $sf(\beta)$ durch Ersetzung von

$(3, NB)(3, NB)$ für $(3, NB)$.

Somit ist $sf(\alpha)$ wk $sf(\beta)$.

"sicher"

Zu einem beliebigen aktiven Ablauf α in Q betrachte $\beta = r(\alpha)$. Die Schaltfolge $sf(\beta)$ entsteht aus $sf(\alpha)$ durch Weglassen einiger Glieder $(3, NB)$. Da NB immer konstant $= \{1, 2, 3\}$ bleibt, stehen die beiden Folgen in der geforderten Relation

$$sf(\beta) \text{ wk } sf(\alpha).$$

Q.E.D.

Auch wenn in diesem Beispiel von Synchronisation noch keine Rede ist, zeigt es immerhin, daß die Begriffe das Erwartete leisten.

Eine weitere Vergrößerung der Relation wk wäre in folgendem Sinne erwägenswert: Die als SNZ beschriebene Klasse unterscheidet zwischen zwei Schaltfolgen, sobald sie an irgendeiner Stelle unterschiedliche Blockaden enthalten. Das entspricht einem Scheduler mit unendlichem Gedächtnis, um sich alle vergangenen Zustände zu merken. Ein endlicher Scheduler entspräche einer Relation wk , die nur endlich viele Äquivalenzklassen erzeugt. Eine solche Äquivalenzklasse kann als Wissen des Schedulers über das bisherige Verhalten des Prozeßsystems aufgefaßt werden; auf die Frage, nach welchen Gesichtspunkten dieses Wissen aus dem gesamten Ablauf abstrahiert wird, gibt es allerdings mehrere Antworten. Eine davon gibt die Betrachtung von V-Systemen in Abschnitt 5.

3.6. Fundamental- und Invarianztheorem

Um mit dem definierten Simulationskonzept Unterschiede zwischen Synchronisationsmechanismen nachweisen zu können, brauchen wir zwei Sätze, die aus Lipton's Theorie ebenfalls bekannt sind (BATH82). Das Invarianztheorem ermöglicht die Zurückführung des Vergleichs auf lokale Betrachtungen, das Fundamentaltheorem wird zu dessen Beweis benutzt. Beide Sätze machen Aussagen über die Simulation bezüglich einer Scheduleklasse, für die noch gewisse Minimalanforderungen gelten müssen.

Schreibweise:

Für eine Schaltfolge

$$\alpha = (p_1, NB_1)(p_2, NB_2) \dots (p_k, NB_k)$$

bezeichne $NB(\alpha) := NB_k$

die Menge der nach α nicht blockierten Prozesse.

Für einen Ablauf α bezeichne

$$NB(\alpha) := NB(sf(\alpha))$$

die Menge der nach dem Ablauf α nicht blockierten Prozesse.

Definition:

Eine Scheduleklasse $M=(M, wk)$ heißt reichhaltig, wenn $M=ZSF_n$ und

$$(\alpha \leq \beta) \Rightarrow (NB(\alpha) \subseteq NB(\beta)),$$

Die Relation wk darf also nicht zu grob sein: eine "schwächere" Folge darf nicht mehr lauffähige Prozesse hinterlassen.

Definition:

Ein Prozeßsystem P heißt explizit-blockierend, wenn für jedes $\alpha \in act(P)$, und für jedes $p \in NB(\alpha)$ eine Aktion a existiert, so daß

$$bz(a) = p \quad \text{und} \quad \alpha a \in act(P).$$

D.h. ein nicht blockierter Prozeß kann auch wirklich eine Aktion ausführen.

Wie bereits gezeigt, sind alle F- und V-Systeme explizit-blockierend. Man erwartet diese elementare Eigenschaft eigentlich als selbstverständlich, aber unser Prozeßmodell läßt auch andere Systeme zu, aufgrund der Möglichkeit von Vorbedingungen.

Fundamentaltheorem:

Seien P, Q explizit-blockierende Prozeßsysteme, r eine Realisierung, $K=(M, wk)$ eine reichhaltige Scheduleklasse, und Q simuliere P bezüglich K durch r prompt.

Sei $\beta \in \text{act}(P)$. Sei $NB(\beta) = X$ und seien

$$b_1, b_2, \dots, b_n$$

diejenigen Aktionen in P , die im Zustand $\text{value}(\beta)$ ausführbar sind.

Dann gibt es ein $\alpha \in \text{act}(Q)$ mit

$$NB(\alpha) = X$$

und Aktionen $a_1, a_2, \dots, a_n$ in Q , die im Zustand $\text{value}(\alpha)$ ausführbar sind und so daß gilt

$$r(a_i) = b_i \quad (i=1,2,\dots,n).$$

Beweis:

Wegen $M=ZSF_n$ ist $\text{sf}(\beta) \in M$; daraus folgt wegen Vollständigkeit der Simulation die Existenz eines α' mit

$$r(\alpha') = \beta \text{ und } \alpha' \in \text{act}(Q).$$

Sei α unter allen solchen α' ein maximales, d.h. jede aktive Fortsetzung von α hat ein von β verschiedenes r -Bild. Daß es ein maximales gibt, folgt aus der Promptheit der Simulation - sonst wären beliebig lange unbeobachtbare Abläufe möglich.

Nun ist wegen $M=ZSF_n$: $\alpha \in M$, daher wegen Sicherheit der Simulation: $\text{sf}(r(\alpha)) \text{ wk } \text{sf}(\alpha)$, also

$$\text{sf}(\beta) \text{ wk } \text{sf}(\alpha).$$

Nach der Definition von "reichhaltig" ist also

$$X \subset NB(\alpha).$$

Sei nun andererseits $p \in NB(\alpha)$, d.h. p die Nummer eines nach α nicht blockierten Prozesses. Nach Voraussetzung (explizitblockierend) gibt es dann eine Aktion a in Q , mit $\text{bz}(a)=p$, so daß

$$\alpha a \in \text{act}(Q).$$

Wegen Maximalität von α kann aber a nicht unbeobachtbar sein, also gilt

$$r(a) = b_i \text{ für ein } i=1,\dots,n;$$

$$r(\alpha a) = \beta b_i, \text{ wo } \text{bz}(b_i)=\text{bz}(a)=p \text{ wegen Struktur-erhaltung.}$$

Folglich ist auch nach β der Prozeß Nummer p in P nicht blockiert, d.h. $p \in X$.

Das gleiche gilt für jeden nach α lauffähigen Prozeß p , und für jeden solchen erhält man eine Aktion a_i , $r(a_i) = b_i$.
Somit ist auch gezeigt $X = NB(\alpha)$.
Q.E.D.

Bemerkung:

Die Forderung nach "Reichhaltigkeit" tritt hier anstelle der Eigenschaft "blockadefrei" in (LIPT74). Sie garantiert einen ausreichenden Zusammenhang der Abläufe in P und Q .

Das Invarianztheorem soll einen lokalen Vergleich von Prozeßsystemen ermöglichen. Dazu muß zunächst gesagt werden, was wir unter lokalem Verhalten verstehen wollen. Die Idee ist dabei, zu fragen, wie sich ein Prozeßsystem verhält, wenn die gerade anstehenden Einzelaktionen aller nicht blockierten Prozesse in unterschiedlichen Reihenfolgen ausgeführt werden. In Lipton's Theorie betrachtet man dabei die Menge der möglichen Reihenfolgen, eine solche (endliche) Menge heißt Schnitt. In einem explizit-blockierenden Prozeßsystem muß man den Begriff etwas anders definieren, weil gilt:

Sind in einem Zustand die Aktionen $a_1, a_2, \dots, a_k$ einzeln ausführbar, dann ist auch die Folge

$$a_1 a_2 \dots a_k$$

und jede Permutation davon ausführbar; das folgt aus der schlichten Tatsache, daß sich die Prozesse erst nach einer Aktion blockieren.

So kommt man zu folgender

Definition:

Ein Schnitt X ist eine Abbildung

$$X \in \text{Abb}(\text{Perm}(n), \text{Pow}(n)),$$

die jeder Permutation der Zahlen $1, 2, \dots, n$ eine Teilmenge der Menge $\{1, 2, \dots, n\}$ zuordnet.

Ein Prozeßsystem P definiert den Schnitt X , wenn es einen aktiven Ablauf β und Aktionen $b_1, b_2, \dots, b_n$ gibt, so daß

$$\beta a_i \text{ ist ebenfalls aktiv für } i = 1, 2, \dots, n;$$

und für jede Permutation $p \in \text{Perm}(n)$ gilt:

$$NB(\beta a_{p(1)} a_{p(2)} \dots a_{p(n)}) = X(p);$$

d.h. X beschreibt für jede Reihenfolge der Aktivierung von $b_1 \dots b_n$

die Menge der danach nicht blockierten Prozesse.

Anmerkung: Im Gegensatz zu (LIPT74) unterscheiden wir hier nicht zwischen "P definiert X" und "P definiert X implizit".

Beispiel

Das folgende Prozeßsystem hat in jedem der drei Prozesse nur eine Aktion.

```
P:  integer x,y init 0;
    process P1 :  do x:= y:= 0 wait x=y=1;
    process P2 :  do x:=1;
    process P3 :  do y:=1;
```

P definiert folgenden Schnitt X:

$X(123) = \{1,2,3\},$	$X(132) = \{2,3\},$
$X(213) = \{2,3\},$	$X(231) = \{2,3\},$
$X(312) = \{2,3\},$	$X(321) = \{2,3\};$

Dabei bezeichnet 132 die Permutation

$p(1)=1, p(2)=3, p(3)=2$ etc.

Für das Invarianztheorem brauchen wir noch etwas strengere Voraussetzungen als für das Fundamentaltheorem. Das folgende Beispiel zeigt, warum die Forderung "reichhaltig" im obigen Sinn noch nicht ausreicht.

Beispiel:

```
P: integer x init 0;
    process P1: (a1:do x:=0 wait x=1;)
    process P2: (a2:do x:=1;)

Q: integer x init 0;
    process Q1: (b11:do x:=0;)(b12:do skip wait x=1;)
    process Q2: (b2:do x:=1;)
```

$r(b11)=a1, r(b12)=\text{lambda}, r(b2)=a2.$

Hier ist r eine Realisierung von P durch Q, und die aktiven Abläufe sind in P:

a1,
a2,
a1 a2,
a2 a1,

und in Q

b11, b11 b12, b11 b12 b2,
b11 b2, b11 b2 b12,
b2, b2 b11, b2 b11 b12.

mit

$r(b11)$	$= a1,$	$NB(b11)$	$= \{1,2\},$
$r(b11 b12)$	$= a1,$	$NB(b11 b12)$	$= \{2\} = NB(a1),$
$r(b11 b12 b2)$	$= a1 a2,$	$NB(b11 b12 b2)$	$= \{1,2\} = NB(a1 a2),$
$r(b11 b2)$	$= a1 a2,$	$NB(b11 b2)$	$= \{1,2\} = NB(a1 a2),$
$r(b11 b2 b12)$	$= a1 a2,$	$NB(b11 b2 b12)$	$= \{1,2\} = NB(a1 a2),$
$r(b2)$	$= a2,$	$NB(b2)$	$= \{1,2\},$
$r(b2 b11)$	$= a2 a1,$	$NB(b2 b11)$	$= \{1,2\},$
$r(b2 b11 b12)$	$= a2 a1,$	$NB(b2 b11 b12)$	$= \{2\} = NB(a2 a1);$

Sei nun K diejenige Scheduleklasse, die durch

$M = ZSF_n,$

$\alpha \text{ wk } \beta \iff NB(\alpha) \subset NB(\beta)$

definiert wird, also unter den reichhaltigen die größte.

Dann ist r vollständig und sicher bezüglich K, denn die Tabelle zeigt:

(vollständig:) es gibt zu jedem $\alpha \in \text{act}(P)$ ein $\beta \in \text{act}(Q)$

mit $NB(\beta) \subset NB(\alpha)$ (sogar $NB(\beta) = NB(\alpha)$!),

(sicher:) für jedes $\beta \in \text{act}(Q)$: $NB(r(\beta)) \subset NB(\beta),$

hier gilt allerdings nicht immer Gleichheit.

Welche Schnitte definieren nun aber diese beiden Prozeßsysteme ?

P definiert den Schnitt X:

$X(12) = \{1,2\}, \quad X(21) = \{2\}.$

Q definiert diesen Schnitt nicht; für β aus der Definition kommen nur in Frage: $\beta = \lambda$ oder $\beta = b11.$

Im ersten Fall muß man die anfangs zur Ausführung anstehenden Aktionen, b11 und b2 betrachten: diese führen zu keiner Blockade, unabhängig von der Ausführungsreihenfolge.

Im zweiten Fall muß man die Aktionen b_{12} und b_2 betrachten; wenn diese beide in irgendeiner Reihenfolge ausgeführt wurden, ist ebenfalls kein Prozeß blockiert.

Wir halten also fest: Q simuliert P bezüglich K , aber definiert nicht jeden Schnitt, den X definiert. Der Grund dafür ist, daß in Q der Prozeß Q_1 vorübergehend noch nicht blockiert sein muß, obwohl in P der zugeordnete Prozeß P_1 in dem (mittels r) entsprechenden Zustand bereits blockiert ist. Das wiederum ist dann möglich, wenn in Q_1 noch Buchführungsaktionen anstehen.

Um das Invarianztheorem zu beweisen, müssen wir also noch strengere Forderungen an die Scheduleklasse K stellen.

Invarianztheorem:

Seien P, Q explizit-blockierende Prozeßsysteme, r eine Realisierung, $K=(M, wk)$ eine Scheduleklasse, so daß gilt:

$M = ZSF_n$ und

$\alpha \text{ wk } \beta \Rightarrow NB(\alpha) = NB(\beta)$.

(Dann ist K insbesondere reichhaltig).

Q simuliere P bezüglich K durch r prompt.

P definiere den Schnitt X .

Dann definiert Q den Schnitt X .

Bemerkung:

Ein Beispiel einer Klasse, die die Voraussetzungen erfüllt, ist SNZ .

Beweis des Invarianztheorems:

Sei β ein aktiver Ablauf in P und seien $b_1, b_2, \dots, b_n$ diejenigen Aktionen in P , die die Definition von " P definiert den Schnitt X " verlangt.

Sei $\alpha \in \text{act}(Q)$, so daß $r(\alpha) = \beta$ und

$\alpha \text{ } a_i \in \text{act}(Q), \quad i=1, 2, \dots, n,$

$r(a_i) = b_i$.

Solche $\alpha, a_1, a_2, \dots, a_n$ existieren nach dem Fundamentaltheorem.
 Für jede Permutation $p \in \text{Perm}(n)$ ist dann nach der Voraussetzung
 über K :

$$\begin{aligned} \text{NB} (\alpha \ a(p(1)) \ a(p(2)) \ \dots \ a(p(n))) &= \\ \text{NB}(r(\alpha \ a(p(1)) \ a(p(2)) \ \dots \ a(p(n)))) &= \\ \text{NB} (\beta \ b(p(1)) \ b(p(2)) \ \dots \ b(p(n))) &= \quad X(p). \end{aligned}$$

Also definiert Q den Schnitt X , nämlich mittels der Aktionen $a_1 \dots a_n$.

Q.E.D.

Der Beweis ist einfach, weil die Voraussetzungen über K recht streng
 sind.

Die Voraussetzung $M=ZSF_n$ geht dabei wesentlich ein; sie ermöglicht es
 nämlich, daß der Simulationsbegriff Aussagen über die volle Menge der
 aktiven Folgen zuläßt, statt diese von vorneherein einzuschränken. Die
 Klasse ZF ("zeitfrei") ist dafür z.B. nicht geeignet, denn dort wären
 Abläufe wie

$$\alpha \ a(p(1)) \ a(p(2)) \ \dots \ a(p(n))$$

i. a. gar nicht zulässig, da Prozeßwechsel nicht jederzeit möglich ist.

4. Anwendungen

In diesem Abschnitt untersuchen wir eine Reihe von Synchronisationsmechanismen, die sich als F-Systeme beschreiben lassen. Diese werden auf bekannte Synchronisationsprobleme angewandt und mittels des Simulationsbegriffs miteinander verglichen. Wir ordnen sie schließlich in eine Hierarchie ein.

Die bisherige Definition von F-Systemen läßt als synchronisierende Aktionen alle Operationen der Form

do A wait B

zu, mit beliebiger Aktion A und beliebiger Bedingung B. Diese Allgemeinheit ist für die Modellbildung gerechtfertigt, für Anwendungen aber zu unstrukturiert. So wie Lipton sich auf die Untersuchung von Mechanismen mit ganzzahligen Semaphoren beschränkt, wollen wir auch hier eine bestimmte Struktur der synchronisierenden Aktionen und der von ihnen manipulierten Daten festlegen.

Vor der Definition dieser Struktur sei hier aber noch ein Beispiel gegeben, wie ein F-System ohne solche Restriktionen aussehen kann. Das Beispiel ist schon aus Abschnitt 2.2 bekannt, nämlich die Operationen K00R+/- . Sie wurden in 2.2. informal präsentiert und sollen jetzt in den Rahmen "F-System" eingepaßt werden.

Dazu benötigen wir wieder für jeden Semaphor s eine Zählvariable

integer $s.ct$

und eine Warteschlange

queue $s.q$,

und nehmen an, für den Datentyp $queue$ stünden die bekannten Operationen $enqueue$ und $tail$ zur Verfügung, außerdem ein Prädikat

$ising(q, x)$,

welches aussagt, ob x als Element in der Warteschlange q enthalten ist.

Dann lassen sich die Operationen K00R+/- wie folgt darstellen:

```

KOOR-(s) :    do (s.ct := s.ct - 1;
                if s.ct < 0
                then enqueue(s.q, ownpid)
                )
            wait not isinq(s.q, ownpid) ;

KOOR+(s) :    do (s.ct := s.ct + 1;
                if s.ct ≤ 0
                then s.q := tail(s.q)
                );

```

Diese Darstellung ist einer Implementierung sehr nahe, außerdem läßt sie offen, wie die Prozedur enqueue verstanden wird, d.h. nach welcher Strategie die Warteschlange verwaltet wird.

Offensichtlich kann man in ähnlicher Weise eine große Klasse von Synchronisationsmaßnahmen darstellen, nämlich alle, die sich mit Warteschlangen beschreiben lassen. Man erhält dabei immer Fortsetzungsbedingungen der Form

not isinq(q, ownpid).

Umgekehrt ist es sogar möglich, jedes F-System mittels Warteschlangen zu implementieren, falls man auf diesen genügend allgemeine Operationen zuläßt. Man wird dann bei jeder Aktion, die potentiell eine andere deblockieren könnte, alle Fortsetzungsbedingungen von wartenden Aktionen prüfen und die betreffenden Aktionen aus der Warteschlange entfernen. Das bedeutet nichts anderes als explizites Deblockieren von Prozessen und ist - wie ja gerade die Allgemeinheit und Implementierungsnähe zeigen - ein sehr unstrukturierter Ansatz.

Statt dessen untersuchen wir Mechanismen von einer Gestalt, die sich in verschiedenen Beispielen als ebenfalls recht vielseitig erwiesen hat, dabei aber ohne Warteschlangen und Mengen auskommt, sondern nur integers als Datenbasis verwendet. Obwohl diese Wahl erst aus den Beispielen hinreichend motiviert wird, sei hier die Definition vorweggestellt.

Definition:

Ein F-System P heit vom Zhler-Typ, wenn es folgende Bedingungen gengt:

1. Die Wertemenge V_a der eigentlichen Variablen zerfllt in drei Komponenten

$$V_a = V_{a1} \times Z^k \times (Z^k)^n,$$

wobei V_{a1} den fr die Synchronisation unbedeutenden Anteil darstellt, Z die Menge der ganzen Zahlen; die Komponente

$$Z^k$$

entspricht k ganzzahligen Synchronisationsvariablen;

davon gibt es fr jeden der n Prozesse eine weitere Kopie.

Ein typisches Element von V_a sei

$$G = (G1, z, (y1, y2, \dots, yn)), \quad z, y_i \in Z^k$$

2. Die synchronisierenden Aktionen von P haben die Form

$$f: \text{ do } (y_i := w(z); z := w'(z)) \text{ wait } z \geq y_i;$$

mit $i = bz(f)$.

Dabei sind die Funktionen

$$w, w' \in \text{Abb}(Z^k, Z^k), \quad (w \text{ linear, und } w'(z) \geq z)$$

von der Aktion f abhngig und definieren diese, zusammen mit der Fortsetzungsbedingung $z \geq y_i$.

3. Keine nicht-synchronisierende Aktion manipuliert die Gren

$$z, y1, y2, \dots, yn.$$

Anschaulich bedeutet das:

- ein Proze erhht die "Zhler" z und wartet dann, bis diese einen Wert y_i erreichen,
- die Zhler werden niemals erniedrigt,
- die Werte y_i sind als lokale Gren des i -ten Prozesses anzusehen.

In typischen Anwendungen ist $w'(z) = z + 1$, dann geben die Zhler an, wie oft bestimmte Aktionen durchlaufen wurden - daher die Bezeichnung - dies trifft jedoch nicht immer zu, da fr w' auch andere nichtverkleinernde Funktionen in Frage kommen.

Die folgenden Abschnitte stellen unterschiedliche Prozesysteme vom Zhlertyp vor.

4.1. Erzeuger-Verbraucher-Probleme

Verschiedene Varianten des Erzeuger-Verbraucher-Problems werden in der Literatur beschrieben. Sie' sollen hier als erste Beispiele für Mechanismen vom Zählertyp dienen.

Allgemein handelt es sich dabei um die folgende Situation:

Ein Prozeß - der Erzeuger oder Produzent - erzeugt als Output eine Folge von Datenelementen, die von einem anderen Prozeß - dem Verbraucher - weiterverarbeitet werden. Dazu werden die Daten in irgendeiner Form zwischengepuffert.

Da man von allen anderen Aktivitäten der Prozesse absieht, nimmt man an, jeder bestünde aus einer Schleife, in der jeweils eine Einheit erzeugt oder verbraucht wird.

Der Verbraucher soll immer dann und nur dann aktiv werden, wenn ein Auftrag für ihn bereit, d.h. der Puffer nicht leer ist.

Der einfachste Fall ist nun das

1-Erzeuger-1-Verbraucher-Problem:

Es wird von folgendem F-System beschrieben:

P1E1V:

```

var s: record e,v :integer end;
    initially s.e=s.v=0;
process ERZE: loop ...
                erzeuge X;
                PROD(s);
            endloop;

process VERB: integer y;
            loop CONS(s);
                verarbeite X;
                ...
            endloop;

```

mit

```

PROD(s) = do s.e:=s.e+1 ;
CONS(s) = do (s.v:=s.v+1; y:=s.v) wait (s.e ≥ y);

```

Anmerkungen:

1. Die record-Schreibweise von Pascal deutet hier an, daß es sich logisch um eine Variable handelt, die durch zwei integers repräsentiert wird. Die Prozesse dürfen selbstverständlich die Komponenten von s nicht in anderer Weise manipulieren als durch die Operationen PROD und CONS. Als handelt sich also um einen abstrakten Datentyp mit diesen beiden Operationen. Diese Datenabstraktion ist sehr wichtig, steht aber hier nicht im Zentrum des Interesses. Auch die Frage, welche Sprachmittel dieser Abstraktion am ehesten gerecht werden, steht hier nicht zur Diskussion.

2. Es dürfte klar sein, wie die Variablen des Beispiels auf die allgemeine Struktur eines F-Systems vom Zählertyp abzubilden sind: Die Werte $s.e$ und $s.v$ sind die beiden globalen Zähler (also $k=2$), die lokale Größe y ist nur im Prozess VERB relevant, sie hat hier nur eine Komponente statt derer k wie in der allgemeinen Definition. Da man zusätzlich eine triviale Komponente mitführen könnte, macht das keinen Unterschied. Die Abbildung w aus der Definition ist ebenfalls nur bei der Aktion CONS relevant und hat die Gestalt

$$w(s.e, s.v) = s.v \quad ;$$

3. Hier wird ein unendlicher Puffer angenommen; der Erzeuger darf beliebig weit "vorausseilen", also beliebig viele Einheiten auf Vorrat produzieren.

4. Es wird nicht behauptet, die angegebene Lösung unterscheide sich wesentlich von der bekannten mit P/V-Operationen ($V(s)$ statt PROD(s), $P(s)$ statt CONS(s)). Vielmehr soll sie nur als erstes Beispiel für die Klasse der Systeme "vom Zählertyp" dienen.

Die Entsprechung zwischen PROD/CONS und P/V wird später in allgemeinerem Rahmen untersucht (Abschnitt 4.3.).

5. Ein "Beweis", daß das angegebene Prozeßsystem das 1-Erzeuger-1-Verbraucher-Problem löst, ist weder nötig noch möglich, weil dieses Prozeßsystem hier die erste formale Beschreibung des Problems selbst ist. Allerdings kann die Funktion in einigen Sätzen erläutert werden:

Die Variablen $s.e$ bzw. $s.v$ geben zu jedem Zeitpunkt an, wie oft bisher die Aktion $PROD(s)$ bzw. $CONS(s)$ aufgerufen wurde; anders ausgedrückt, $s.e$ ist die Anzahl der erzeugten Einheiten, $s.v$ die Anzahl der Einheiten, deren Verarbeitung begonnen wurde. Letztere kann um 1 größer sein als die Anzahl der tatsächlich verarbeiteten Einheiten. In der lokalen Größe y wird der Wert von $s.v$ zwischengespeichert. Da $s.v$ an keiner anderen Stelle verändert wird, könnte man auf y verzichten; es wurde eingeführt, um der allgemeinen Form zu genügen.

Die Fortsetzungsbedingung " $s.e \geq y$ " drückt nun aus, daß mindestens eine Einheit mehr produziert als konsumiert wurde.

6. Es fällt eine enge Verwandtschaft zu Habermann's Semaphor-Invariante auf (HABE72): Diese zählt ja auch die Aufrufe P und V , und unterscheidet bei der P -Operation die Anzahl der aufgerufenen ("execute") und die Anzahl der fortgesetzten ("pass") Operationen. In der Situation des Beispiels gilt

$$\begin{aligned} s.e &= s.v - 1 && \text{g.d.w. der Verbraucher ist blockiert,} \\ s.e &\geq s.v && \text{sonst.} \end{aligned}$$

D.h. bezeichnet man die Anzahl der fortgesetzten $CONS$ -Operationen mit p , so gilt

$$p = \min(s.v, s.e).$$

Tatsächlich sind die angegebenen Operationen $PROD$ und $CONS$ die Entsprechung der altbekannten P/V -Operationen für F -Systeme, und die zitierte Arbeit Habermann's zeigt, daß schon die damaligen Überlegungen eher auf den von uns als F -System bezeichneten Ansatz hin ausgerichtet waren.

7. Das Beispiel behandelt nur den für die Koordinierung relevanten Teil des Problems; insbesondere wurden die eigentlichen Zugriffe zu dem gemeinsamen Puffer nicht berücksichtigt.

n-Erzeuger-1-Verbraucher-Problem

Der angegebene Mechanismus überträgt sich ohne weiteres auf den Fall von mehreren Erzeugerprozessen, die einen Verbraucher beliefern, wie es etwa auf die Situation einer Ausgaberroutine zutrifft, die für verschiedene Anwenderprozesse tätig ist. Dabei wird gefordert: jede erzeugte Informations-Einheit muß verarbeitet werden. Das erreichen die Operationen PROD und CONS.

Erstes 1-Erzeuger-n-Verbraucher-Problem

Eine Möglichkeit der Verallgemeinerung auf der Verbraucherseite besteht darin, mehrere gleichartige Verbraucher parallel arbeiten zu lassen, und zu verlangen: jede erzeugte Einheit wird von genau einem Verbraucher verarbeitet. Das entspricht etwa der Situation eines Stapelbetriebs mit mehreren gleichen Ausgabedruckern: Jeder Job soll auf genau einem Drucker ausgegeben werden, egal auf welchem. Die Parallelität erhöht dabei den Durchsatz.

Auch diese Aufgabe beschreiben die Operationen PROD und CONS korrekt. Im Unterschied zum 1-Verbraucher-Fall wird dabei die Variable s.v von mehreren Prozessen verändert, und ihr lokales Abbild y ist nicht mehr entbehrlich. Es zeigt sich jetzt auch ein Unterschied zu P/V: warten mehrere Prozesse, und ein PROD wird ausgeführt, dann wird zwangsläufig genau der Verbraucher-Prozeß fortgesetzt, der am längsten wartet, denn nur für sein lokales y ist die Bedingung " $s.e \geq y$ " erfüllt.

Beachte: keine zwei Prozesse haben das gleiche y, weil s.v jedesmal erhöht wird.

Zweites 1-Erzeuger-n-Verbraucher-Problem

Man kann aber die Parallelität auf der Verbraucherseite auch anders auffassen:

Verschiedene Verbraucherprozesse sollen verschiedene Aufgaben erfüllen, und es wird gefordert, daß jede erzeugte Einheit von jedem Verbraucher verarbeitet wird. Als Anwendung läßt sich ein Meldungssystem denken, bei dem jede erzeugte Meldung auf verschiedenen Geräten gedruckt und zusätzlich in einer Datei archiviert werden soll.

Hierzu definieren wir ein neues Paar von Operationen; sie arbeiten auf folgendem Datentyp:

```
var s : record  e: integer; ,
                v: array(1..n) of integer;
            end;
```

```
PRODP(s) =  do s.e := s.e+1;
```

```
CONSP(s) =  do (y := s.v(ownpid) := s.v(ownpid) + 1)
            wait (s.e  $\geq$  y);
```

Durch die Vervielfachung von s.v zu einem Feld wird erreicht, daß die einzelnen Komponenten je einem Prozeß als privat zugeordnet sind; die Verbraucher beeinflussen sich gegenseitig nicht.

2-Erzeuger-2-Verbraucher-Problem

Von etwas speziellerer Natur ist die folgende Aufgabenstellung nach {HOFM82):

Ein Erzeuger E1 schreibt in den Puffer P1, aus dem der Verbraucher V1 liest;

Ein Erzeuger E2 schreibt in den Puffer P2, aus dem der Verbraucher V2 liest;

V1 hat dabei Priorität vor V2, d.h. V2 darf nur aktiv werden, wenn der Puffer P1 leer ist.

Dieses Problem läßt sich wohl als F-System, nicht aber im Zähler-Typ beschreiben:

```
PROD1 =  do s1.e := s1.e +1;
CONS1 =  do s1.v := s1.v +1 wait (s1.e  $\geq$  s1.v);

PROD2 =  do s2.e := s2.e +1;
CONS2 =  do s2.v := s2.v +1
            wait (s2.e  $\geq$  s2.v and s1.v > s1.e);
```

Das zweite Konjunkt in der Fortsetzungsbedingung für CONS2 läßt sich nicht in die Form des Zählertyps bringen, weil die Größe $s1.e$ darin als dynamische Größe vorkommt, d.h. es genügt nicht, den Wert zum Zeitpunkt der Ausführung von CONS2 festzuhalten, sondern es müssen jeweils die aktuellen Werte von $s1.v$ und $s1.e$ verglichen werden. Darin drückt sich gleichzeitig die Eigenschaft der Aufgabenstellung aus, nicht fair zu sein: bei fortgesetzter Aktivität der Prozesse E1/ V1 wird nämlich V2 "ausgehungert", er bleibt blockiert.

Dieses negative Beispiel lehrt :

mit F-Systemen vom Zählertyp läßt sich faires Verhalten im Sinn von "Vermeidung von Aushungern" leichter beschreiben als das Gegenteil.

Wir werden darauf zurückkommen (4.3.2.).

Hier also die faire Version der obigen Aufgabe:

(Statt des ganzen Prozeßsystems werden wieder nur die synchronisierenden Aktionen angegeben)

```

PROD1 =  do  $s1.e := s1.e+1$ ;
CONS1 =  do ( $y := s1.v := s1.v + 1$ ) wait ( $s1.e \geq y$ );

PROD2 =  do  $s2.e := s2.e+1$ ;
CONS2 =  do ( $y2 := s2.v := s2.v+1$ ;
            $y1 := s1.e+1$ )
           wait ( $((s2.e, s1.v) \geq (y2, y1))$ );

```

Die letzte Bedingung ist komponentenweise zu verstehen und sagt aus:

$s2.e \geq y2$ - d.h. Puffer P2 ist nicht leer, und

$s1.v \geq y1$ - d.h. alle in Puffer P1 vor Aufruf von CONS2 abgelegten Elemente sind inzwischen verarbeitet. Der Puffer muß nicht leer sein.

Man hat also eine schwächere Art von Priorität des Prozesses V1 über den Prozeß V2.

4.2. Ungepufferte Koordinierung

Bei einer Reihe von Anwendungen sollen die freigebenden Operationen (wie $\text{PROD}(s)$, $\text{V}(s)$) nicht wie im vorigen Abschnitt gepuffert werden, sondern es genügt, einen Prozeß fortzusetzen, falls er wartend (blockiert) ist, und nichts zu tun, falls er bereits tätig ist. Diese Art von Koordinierung bezeichnen wir als ungepuffert. Sie wird in Betriebssystemen oft durch einfache START-Aufrufe realisiert; ein bereits gestartetes Programm zu starten, hat dabei keine Wirkung, man sagt der Start "geht verloren".

4.2.1. sleep/wakeup

Das einfachste Beispiel für ungepufferte Koordinierung sind die Operationen sleep und wakeup; sie wurden schon in 2.1. beschrieben; die folgende Darstellung ist zwar etwas komplizierter, entspricht aber dem Schema des Zählertyps:

```
var s: record e,v: integer end;

sleep(s) := do (y:=s.v:= s.e+1) wait s.e ≥ y;

wakeup(s) := do s.e:= s.e+1;
```

Hier wird $s.e$ wie bisher schrittweise erhöht, $s.v$ dagegen kann Werte überspringen. Die Wahl $y:= s.e+1$ hat zur Folge, daß "sleep" genau beim nächsten "wakeup" fortgesetzt wird. Die Größe $s.v$ selbst wäre hier entbehrlich, wird aber der Einheitlichkeit wegen beibehalten.

Man überzeugt sich auch leicht, daß $s.v$ nie erniedrigt wird, wie es die Definition verlangt: es gilt nämlich jederzeit

$$s.v \leq s.e+1,$$

somit kann die Zuweisung

$$s.v:= s.e+1$$

die Variable $s.v$ nicht erniedrigen.

4.2.2. Aktualisierungsprobleme

Die in diesem Abschnitt diskutierte Klasse von Aufgabenstellungen spielt in der Praxis ebenfalls eine wichtige Rolle, wurde aber meines Wissens in der Literatur unter dem Gesichtspunkt der Prozeßkoordination noch nicht betrachtet.

In einer Echtzeit-Anwendung gibt es ein "Prozeßabbild", etwa eine Datei, die jederzeit die relevanten Daten über den vom Rechner gesteuerten oder überwachten Teil der Außenwelt enthält.

Dieses wird von einer Reihe von Programmen - den Erfassungsprogrammen - erfaßt und abgespeichert, etwa indem jeweils eine Gruppe von Meßwerten von einem Programm betreut wird.

Eine weitere Gruppe von Programmen hat die Aufgabe, das jeweils aktuelle Prozeßabbild auszugeben; darunter kann typischerweise eine Darstellung an Bildschirmen verstanden werden. Daher nennen wir diese Programme Anzeigeprogramme.

Die sich daraus ergebende Koordinierungsaufgabe läßt sich in erster Näherung so beschreiben:

Ein Anzeigeprogramm soll immer dann aktiv werden, wenn sich das Prozeßabbild geändert hat. Es soll dann die Ausgabe aktualisieren. Dabei ist es nicht erwünscht, daß bei mehreren aufeinanderfolgenden Änderungen die Anzeigeprogramme immer auch mehrfach durchlaufen werden.

Wenn sich ein Zeitverzug aufgrund momentan hoher Auslastung ergibt, soll nämlich nicht jede einzelne Änderung des Prozeßabbilds angezeigt werden, sondern nur jeweils der letzte, also aktuelle Zustand.

Darin unterscheiden sich solche Systeme von denen aus Abschnitt 4.1.: der "Anstoß" zum Aktualisieren der Ausgabe erfolgt nicht gepuffert, sondern "gedächtnislos". Ein mehrfacher Anstoß hat die gleiche Wirkung wie ein einzelner, außer wenn sich das angestoßene Programm inzwischen blockiert.

Man könnte zunächst versuchen, diese Aufgabe mit dem Operationenpaar sleep/wakeup zu lösen, das führt jedoch nicht zum Ziel:

ERF_ANZ0: ----- nicht korrekt -----

```

process ERFASS: loop
    ...
    (Änderung am Prozeßabbild);
    wakeup(s);
endloop;

process ANZEIG: loop
    ...
    (Ausgabe aktualisieren);
    ...
    sleep(s);
endloop;

```

Hier ist nämlich folgender Ablauf möglich:

während ANZEIG gerade die Ausgabe aktualisiert, wird wakeup durchlaufen. Bei der anschließenden sleep-Operation wird ANZEIG blockiert, und es ist nicht garantiert, daß die letzte Änderung des Prozeßabbilds noch berücksichtigt wurde.

Um dem abzuhelpen, muß der Verarbeitungsteil von ANZEIG mit zwei synchronisierenden Aktionen geklammert werden:

```

ERF_ANZ1:
    var s: record e,v: integer end;

process ERFASS: loop
    ...
    (Änderung am Prozeßabbild);
    do s.e:= s.e+1;
endloop;

process ANZEIG: loop
    do {y:=s.v:= s.e+1};
    ...
    (Ausgabe aktualisieren);
    ...
    do skip wait s.e ≥ y;
endloop;

```

Wegen der Zweiteilung der synchronisierenden Aktionen im Prozeß ANZEIG entspricht dieses Beispiel nicht mehr ganz der Definition "Zähler-Typ". Diese Zweiteilung ist nötig, um zu gewährleisten, daß ein Anzeigeprogramm auch dann noch ein weiteres Mal angestoßen wird, wenn es zum Zeitpunkt der Aktion $s.e:=s.e+1$ noch aktiv ist.

Von dieser Zweiteilung abgesehen, ist der angegebene Mechanismus identisch mit sleep/wakeup.

Die Funktion stimmt weitgehend mit der von sleep/wakeup überein, mit dem Unterschied, daß ein wartendes Anzeigeprogramm genau dann fortgesetzt wird, wenn seit Beginn der Anweisungsfolge "Ausgabe aktualisieren" eine Änderung am Prozeßabbild vorgenommen wurde.

Zur Funktionsweise muß noch angemerkt werden: da die Größe y als lokal angesehen wird, können mehrere Anzeigeprogramme die Synchronisationsoperationen verwenden, ohne einander zu beeinflussen. Die Fortsetzungsbedingung eines Prozesses ergibt sich immer nur aus dem Wert von $s.e$ zum Zeitpunkt des Schleifenbeginns.

Die Reihenfolge der Aktionen wurde hier so gewählt, daß beim ersten Durchlauf jedenfalls eine Ausgabe erfolgt, erst die folgenden Durchläufe sind dann ereignisgesteuert.

4.2.3. Erweiterungen

Analog zu den erweiterten Semaphore-Operationen P/V-multiple, P/V-chunk und P/V-general kann man auch sleep/wakeup auf verschiedene Arten verallgemeinern:

sleep/wakeup-multiple

Durch Zulassung mehrerer Semaphore in einer einzigen Operation erhält man folgende Operationen:

```

var s,y : array (1..n) of record e,v: integer end;

sleep_m (A: set of (1..n)) :=
  do ( for (i in A) do s(i).v := s(i).e +1;
      y := s.v
      )
      wait s.e ≥ y;

wakeup_m (A: set of (1..n)) :=
  do for (i in A) do s(i).e := s(i).e +1 ;

```

Anmerkungen zur Schreibweise:

Das Konstrukt

```
for (i in A) do Anweisung;
```

bedeutet eine Laufschleife über die Menge A.

Die Zuweisung und der Vergleich von Vektoren sind immer komponentenweise zu verstehen.

Wirkung dieser Operationen:

Wie beim einfachen sleep/wakeup kann man jede Komponente $s(i)$ als ein Ereignis verstehen. Die Operation `wakeup_m` signalisiert dann den Eintritt einer Menge von Ereignissen. Dagegen bewirkt `sleep_m`, daß sich der Prozeß solange blockiert, bis jedes der angegebenen Ereignisse eingetreten ist.

Ein Anwendungsbeispiel wäre etwa ein Programm, das Unteraufträge an zwei andere Prozesse abgegeben hat und erst nach Fertigstellung beider Aufträge fortgesetzt werden soll.

sleep/wakeup-chunk

Ein Programm soll zum Zwecke der Datensicherung eine Datei kopieren, und zwar bei jeder Änderung der Datei. Dies läßt sich mit dem einfachen sleep/wakeup bzw. mit dem abgewandelten Mechanismus aus 4.2.2. erreichen. Nun kann man aber fordern, die Sicherungskopie soll nicht bei jeder, sondern nur bei jeder hundertsten Änderung erfolgen. Das leistet folgende Variante der Operationen; die Zähler werden dabei um beliebige Stücke (chunk) erhöht:

```

var s : record e,v: integer end;
integer y;
sleep_c (s, k) :=
  do y := s.e + k
  wait s.e ≥ y ;

wakeup_c (s, k) :=
  do s.e := s.e + k;

```

Damit löst folgendes Programm die gestellte Aufgabe:

BEISPIEL:

```

process P1 : loop
  (Datei ändern)
  ...
  wakeup(s,1);
endloop;

process P2 : loop
  sleep(s,100);
  ...
  (Datei kopieren)
endloop

```

sleep/wakeup-general

Wenn man in den obigen Operationen sleep_c/wakeup_c statt einzelner integer-Variablen für s.e, s.v, y und k Vektoren zuläßt, erhält man das Operationenpaar sleep/wakeup-general oder sleep_g/wakeup_g. Es verallgemeinert gleichzeitig die "chunk"- und "multiple"-Versionen.

Weitere Kombinationen

Die bisherigen Beispiele dieses Abschnitts verwenden weniger komplexe Operationen als die Definitionen erlauben würden: insbesondere genügt es oft, die Verallgemeinerung nur auf die sleep-Operation zu beziehen, während "wakeup" nur ein einzelnes Ereignis signalisiert; so kommen wir zu den Mechanismen

sleep_m/wakeup, sleep_c/wakeup, sleep_g/wakeup.

sleep/wakeup-or

Häufiger kommt sogenanntes Mehrfachwarten in einem anderen Sinn vor: ein Prozeß wartet solange, bis wenigstens eines von mehreren Ereignissen eintritt. Dafür gibt es die unterschiedlichsten Beispiele, etwa ein Batch-Job-Handler wartet auf eines der folgenden Ereignisse:

- Fertigstellung eines Jobs,
- Ankunft eines neuen Jobs,
- Operator-Eingriff.

Oder ein Programm der Bedien-Oberfläche (wie in UNIX das sog. shell) wartet auf

- Bedienereingabe,
- Fertigstellung eines Hintergrund-Jobs.

Die entsprechenden Synchronisationsoperationen kann man als Abwandlung von sleep/wakeup so darstellen:

```
var z: array (1..n) of integer;

sleep_or (A: set of (1..n)) :=
  do z := 0
  wait (für ein i ∈ A ist z(i)=1)

wakeup (i) := do z(i) := 1;
```

Sie lassen sich allerdings nicht innerhalb des Zähler-Typs modellieren.

4.2.4. Ungepufferte Starts

Bei den bisher betrachteten ungepufferten Mechanismen gilt:

Mit einer sleep-Operation werden alle auf das entsprechende Ereignis wartenden Prozesse fortgesetzt. Verlangt man statt dessen die Deblokierung von nur jeweils einzelnen Prozessen, so kommt man zu einem Mechanismus, der in (PIEP77, S. 121ff.) in Zusammenhang mit der "Ulmer Lösung" des Philosophenproblems beschrieben wird:

Es gibt zwei Operationen START(s) und P(s), die folgendermaßen zusammenspielen:

Die P-Operation wird - wie üblich - bei nichtpositivem Semaphore in einer Warteschlange blockiert; die START-Operation deblockiert jeweils einen der wartenden Prozesse, falls einer wartet, andernfalls geht der Start verloren ; - dies wurde inspiriert vom Betriebssystem ORG I des Siemens-Rechners 305.

Wenn wir diesen Mechanismus hier aufgreifen, dann nicht, um zu behaupten, er sei besonders vorteilhaft oder logisch. Es soll nur gezeigt werden, welche Variationsmöglichkeiten F-Systeme vom Zählertyp bieten.

Im Gegensatz zu obiger informeller Beschreibung, die offen läßt, welcher von ggf. mehreren wartenden Prozessen gestartet wird, legt unsere Version die Reihenfolge als FIFO fest:

```
P_ULM (s) :=
  do (s.v := max(s.e, s.v) + 1;
      y := s.v)
  wait s.e ≥ y;
```

```
START (s) :=
  do s.e := s.e + 1;
```

Die Bildung des Maximums bewirkt:

falls $s.e > s.v$, dann gehen Starts verloren,

falls $s.v > s.e$, dann warten schon Prozesse, und der aktuelle reiht sich in deren Folge als letzter ein.

Ebenfalls denkbar ist die Variante

```
P_ULM' (s) :=
    do (s.v := max(s.e, s.v+1);
        y := s.v)
    wait s.e ≥ y;
```

Sie bewirkt, daß maximal ein Start gepuffert wird, statt wie oben keiner. Mit dieser Variante ist übrigens sogar das Problem des gegenseitigen Ausschlusses lösbar, nämlich mit Anfangswerten

s.e=1, s.v=0.

Bei paarweiser Verwendung in der Art

```
P_ULM'(s); ..(kritischer Abschnitt).. ; START(s);
```

gilt nämlich immer

s.v ≥ s.e,

und die Operationen verhalten sich daher wie die gewöhnlichen P/V-Operationen, bzw. wie deren F-System-Entsprechung CONS/PROD.

Weiter läßt sich auch hier durch Einführung von Semaphor-Vektoren verallgemeinern:

```
P_ULM_general (s,k) :=
    do (s.v := max(s.e, s.v) + k;
        y := s.v)
    wait s.e ≥ y;
```

```
START_general (s,k) :=
    do s.e := s.e + k;
```

wobei s.e, s.v, y und k Vektoren mit nichtnegativen Komponenten sind.

Die Mechanismen

P_ULM/START-multiple und P_ULM/START-chunk

haben dann die offensichtliche Bedeutung, werden aber nicht weiter untersucht.

4.2.5. Übersicht

Der Vergleich der bisher definierten Mechanismen ergibt für die jeweiligen Basisversionen (ohne chunk, multiple, general) folgende gemeinsame Form

Warteoperation:

```
do (s.v := f(s.e,s.v) + 1;
    y := s.v)
wait s.e  $\geq$  y;
```

Freigabeoperation:

```
do s.e := s.e + 1;
```

Und die Funktion f definiert die Eigenschaften des Mechanismus wie folgt:

$f =$	Name	Bedeutung

s.v	CONS/PROD	echt gepufferte Starts einzelner Prozesse
s.e	sleep/wakeup	ungepufferte Starts aller wartender Prozesse
max(s.e,s.v)	P_ULM/START	ungepufferte Starts einzelner Prozesse.

4.3. Belegungsprobleme

Als Belegungsprobleme bezeichnen wir die Koordinierungsaufgaben, die sich aus der gemeinsamen Benutzung von Betriebsmitteln ergeben: der kritische Abschnitt, in welchem ein Prozeß solche Mittel benötigt, wird geklammert von Synchronisationsoperationen "belegen" und "freigeben". Im Fall eines einzelnen Betriebsmittels lösen die einfachen P/V-Operationen diese Aufgabe; bei mehreren besteht die Gefahr von Verklemmungen, wenn die Belegungen in falscher Reihenfolge erfolgen. Die Forderung

"ein Prozeß muß alle benötigten Betriebsmittel
auf einmal anfordern "

vermeidet diese Verklemmungen. Sie wird mit den Operationen P/V-general erfüllt:

```
semaphore s1,s2,...,sn ;
P(s1,k1;s2,k2;...;sn,kn) :=
    when (s1 ≥ k1 and ... and sn ≥ kn)
    do (s1:= s1-k1; ... sn:= sn-kn);

V(s1,k1;s2,k2;...;sn,kn) :=
    do (s1:= s1+k1; ... sn:= sn+kn);
```

Wenn k_i die Anzahl der benötigten bzw. freizugebenden Exemplare des i -ten Betriebsmittels ist, und wenn die Anfangswerte der Semaphore den anfänglichen Vorrat von jedem Betriebsmittel wiedergeben, dann lösen diese Operationen die Koordinierungsaufgabe, d.h. sie erfüllen folgende Forderungen:

- Verklemmungsfreiheit,
- die Anzahl belegter Exemplare eines Betriebsmittels ist nie größer als der Vorrat.

Was mit diesen Operationen dagegen grundsätzlich nicht erfaßt wird, ist Fairness, oder die Vermeidung des Aushungerns. Man sagt, ein Prozeß wird ausgehungert, wenn er nie aktiv wird, obwohl er nicht dauernd blockiert ist. Üblicherweise umgeht man dieses Problem, indem man vom Schedule die Eigenschaft "Fairness" verlangt; wie aber in Abschnitt 2.4. gezeigt wurde, ist dieses Vorgehen fragwürdig: zumindest die

üblichen Implementierungen von P/V-general lassen Aushungern unabhängig vom Schedule zu !

In der Praxis der Betriebsprogrammierung wurde deshalb eine weitere Lösung für Belegungsprobleme entwickelt (WETT78), der Algorithmus der Bestellvektoren. Er läuft darauf hinaus, daß jede Anforderung, die nicht sofort gewährt werden kann, als Bestellung registriert wird (hier wieder das Grundprinzip für F-systeme: eine Wirkung vor der Blockierung !); bereits bestellte Betriebsmittel können dann nicht an andere Prozesse als den Besteller zugeteilt werden. Dieser einigermaßen komplizierte Algorithmus ist in (WETT78) ausführlich dargestellt und soll hier nicht näher beschrieben werden. Die folgende Lösung als F-System erzielt das gleiche Ergebnis und erscheint mir klarer.

Es ist allerdings einschränkend zu bemerken, daß Wettstein von verschiedenen "dringlichen" Prozessen ausgeht; die Dringlichkeit kann auf beliebige Art festgelegt sein; in unsere Lösung geht nur die zeitliche Reihenfolge ein, d.h. von zwei wartenden Prozessen ist immer der länger wartende der dringlichere.

Definition des Operationenpaars ANFO/FREI-general:

```
var s,y : array (1..n) of record e,v: integer end;
```

```
ANFO_g (k : array (1..n) of integer) :=
  do (s.v := s.v + k; y(*) := s(*).v * sgn(k(*)) )
  wait s.e ≥ y;
```

```
FREI_g (k : array (1..n) of integer) :=
  do s.e := s.e + k;
```

Neben den schon verwendeten Vektoroperationen kommt hier die Signumfunktion sgn vor, sowie die $*$ -Schreibweise für komponentenweise Ausführung der Zuweisung

```
y(i) := s(i).v * sgn(k(i)).
```

k ist der Anforderungsvektor: vom i -ten Betriebsmittel werden $k(i)$ Exemplare angefordert/freigegeben. Daraus ergibt sich die Einschränkung:

```
k(i) ≥ 0 (i=1,..., n).
```

Analog zu den erweiterten P/V-Operationen ist dies die allgemeine Form eines Mechanismus, von dem auch folgende schwächeren Versionen interessant sind:

ANFO/FREI - mit $n = 1$ und $k(1)=1$,

ANFO/FREI-chunk:

ANFO_c/ FREI_c - mit $n = 1$,

ANFO/FREI-multiple:

ANFO_m / FREI_m - mit $k(i)=1 \quad (i=1,\dots,n)$.

Anmerkungen:

Wenn hier als Parameter der Vektor k im ganzen genannt wird, dient das nur der bequemen Schreibweise und hat den gleichen Effekt wie in der obigen Darstellung von P/V-general die Aufzählung aller Komponenten s_i mit $k_i \neq 0$. Letztere Form wäre angemessen in einer Anwendung, wo nur wenige Komponenten von Null verschieden sind. Beide Darstellungen werden ab hier als austauschbar betrachtet, solange es ohne Mißverständnisse möglich ist.

Außerdem bezeichnen wir mit "ANFO/FREI" allgemein alle vier Versionen, solange die Unterscheidung nicht nötig ist.

Aus den Definitionen ist bereits klar, daß die einfache Version ANFO/FREI identisch ist mit den in 4.1. eingeführten Operationen PROD/CONS.

Zur Funktionsweise von ANFO/FREI-general:

In Abschnitt 4.3.2. wird die Verhungerungsfreiheit gezeigt und die Funktionsweise strenger erfaßt. Zum besseren Verständnis hier eine informelle Beschreibung:

1. Die Anfangswerte sind so zu besetzen, daß für alle i gilt:

$s(i).v = 0$, und

$s(i).e = \text{Anzahl der verfügbaren Betriebsmittel.}$

2. Der Wert $s.e$ gibt jeweils die Summe der bisherigen Freigaben plus Anfangswerte an.

3. Der Wert $s.v$ gibt jeweils die Summe der bisherigen Anforderungen an, einschließlich der noch blockierten.

4. Die Verwendung der Signum-funktion bewirkt, daß nur die wirklich angeforderten Betriebsmittel Einfluß auf die Blockierung eines Prozesses haben können. Für die Komponenten mit $k(i)=0$ ist nämlich die Ungleichung $s(i).e \geq y(i)$ immer erfüllt.

5. Ein Prozeß, dessen Anforderung erfüllt werden könnte, kann dennoch blockiert werden, falls ein anderer schon wartet; dies ist genau dann der Fall, wenn die neue Anforderung größer ist als der momentane Bestand minus die vorgemerkten Bestellungen.

6. Der momentane Bestand an einem Betriebsmittel, bei P/V-general ausgedrückt im Wert des betreffenden Semaphores, kommt in diesem Algorithmus nicht direkt zum Ausdruck. Er läßt sich jedoch darstellen als

$$s.e - z,$$

$$z = (\text{Summe aller schon } \underline{\text{gewährten}} = \text{deblockierten Anforderungen}).$$

Auch dies wird in 4.3.2 weiter formalisiert.

Beispiel

Dieses Beispiel soll die Funktionsweise von ANFO/FREI-general verdeutlichen:

```
ANFFRE:  var s : array(1..3) of record e,v : integer end;
         initially s.e=(2,4,1);
```

```
process P1:  ANFO(1,3,0); ... ; FREI(1,3,0);
process P2:  ANFO(0,1,1); ... ; FREI(0,1,1);
process P3:  ANFO(1,2,0); ... ; FREI(1,2,0);
process P4:  ANFO(2,0,0); ... ; FREI(2,0,0);
process P5:  ANFO(1,3,0); ... ; FREI(1,3,0);
```

Die fünf Prozesse konkurrieren um drei Betriebsmittel BM1, BM2, BM3.

Hier ist z.B. folgender Ablauf möglich:

Aktion:	s.e	s.v	s.e-z	Bemerkung
	(2,4,1)	(0,0,0)	(2,4,1)	Anfangswert
(P1: ANFO(1,3,0))	(2,4,1)	(1,3,0)	(1,1,1)	y=(1,3,0)
(P3: ANFO(1,2,0))	(2,4,1)	(2,5,0)	(1,1,1)	y=(2,5,0), P3 blockiert
(P2: ANFO(0,1,1))	(2,4,1)	(2,6,1)	(1,1,1)	y=(0,6,1) P2 ebenfalls blockiert, weil BM2 von P3 vorbestellt - obwohl das angeforderte Exemplar vorhanden ist.
(P1: FREI(1,3,0))	(3,7,1)	(2,6,1)	(1,1,0)	Wegen $(3,7,1) \geq (2,5,0)$ und $(3,7,1) \geq (0,6,1)$ werden jetzt beide Prozesse P3 und P2 deblockiert.
(P4: ANFO(2,0,0))	(3,7,1)	(4,6,1)	(1,1,0)	y=(4,0,0), P4 blockiert
(P5: ANFO(1,2,0))	(3,7,1)	(5,8,1)	(1,1,0)	y=(5,8,0), P5 blockiert
(P2: FREI(0,1,1))	(3,8,2)	(5,8,1)	(1,2,1)	P5 bleibt blockiert, weil BM1 von P4 vorbestellt - obwohl sein Bedarf (1,2,0) vorhanden ist.
(P3: FREI(1,2,0))	(4,10,2)	(5,8,1)	(0,4,1)	Wegen $(4,10,2) \geq (4,0,0)$ wird P4 deblockiert, P5 aber nicht.
(P4: FREI(2,0,0))	(6,10,2)	(5,8,1)	(1,2,1)	Wegen $(6,10,2) \geq (5,8,0)$ wird P5 deblockiert.
(P5: FREI(1,2,0))	(7,12,2)	(5,8,1)	(2,4,1)	

4.3.1. Das Philosophenproblem

Dieses bekannte Problem geht auf Dijkstra (DIJK72) zurück und wurde in der Literatur häufig erörtert.

Fünf Philosophen essen an einem Tisch; zwei Tischnachbarn verfügen gemeinsam nur über eine Gabel. Da jeder zum Essen (auf der Speisekarte stehen Spaghetti) zwei Gabeln benötigt, können zwei Tischnachbarn nie gleichzeitig essen.

Dieses Problem paßt in den Rahmen der Belegungsprobleme: der gegenseitige Ausschluß zweier Esser rührt von der gemeinsamen Benutzung des Betriebsmittels "Gabel" her. Wir wollen also die Operationen ANFO/FREI auf diese Aufgabe anwenden.

Wenn $GB(i)$ den Bedarfsvektor des i -ten Philosophen bezeichnet, dann gilt:

$$\begin{aligned} GB(1) &= (1,1,0,0,0), \\ GB(2) &= (0,1,1,0,0), \\ GB(3) &= (0,0,1,1,0), \\ GB(4) &= (0,0,0,1,1), \\ GB(5) &= (1,0,0,0,1). \end{aligned}$$

Damit läßt sich der Tagesablauf eines Philosophen so beschreiben: (ANFO/FREI steht hier für ANFO/FREI-multiple, da die Anforderungsvektoren keine Komponenten größer 1 haben)

```
process PHILO(i) :
    loop      ...(denken) ...
              ANFO( GB(i) );
              ...(essen) ...
              FREI( GB(i) );
    endloop;
```

Der Anfangswert der Zählvariablen muß dabei

$$s.e = (1,1,1,1,1)$$

betragen, entsprechend den fünf Gabeln.

Gegenüber den bekannten Lösungen ist festzustellen:

1. Hier kann kein Philosoph ausgehungert werden wie bei der Verwendung von P/V-multiple; das wird im nächsten Abschnitt gezeigt.

2. Die Lösung kommt dennoch ohne einen besonderen Zustand "sehr hungrig" aus, ja überhaupt ohne explizite Zustandsvariablen, wie sie in (HOFM74) ("Erlanger Lösung") und (PIEP77) vorgeschlagen wurden.

3. Die Vermeidung des Aushungerns verlangt, daß in gewissen Fällen ein Philosoph am Essen gehindert wird, obwohl beide von ihm benötigten Gabeln frei sind. Wann dies jedoch genau geschieht, darauf gibt es mindestens zwei verschiedene Antworten, die auch unterschiedliche Abläufe bewirken, aber die beide zum Ziel der Fairness führen. Es muß auf die Auffassung hingewiesen werden, daß ein Synchronisationsproblem im strengen Sinn nur angegeben werden kann durch ein Prozeßsystem, das man als Lösung akzeptiert (HOFM82). In diesem Sinn wäre die Forderung "kein Aushungern" noch nicht exakt genug, weil sie nicht festlegt, wie lange ein Philosoph hungern darf, bevor er zum Essen zugelassen werden muß.

Die vorliegende Lösung wählt diese Zeit sehr kurz: sobald ein Philosoph überhaupt hungrig ist (= die Anforderung ANFO gestellt hat), darf schon sein später zu Tisch gekommener Nachbar nicht mehr zum Essen zugelassen (= deblockiert) werden. Sobald dann der evtl. früher gekommene Nachbar fertig ist, bekommt der hungrige dessen, und dazu die "bestellte" Gabel.

Das Beispiel eines Ablaufs soll wieder den Mechanismus verdeutlichen:

Aktion	s.e	s.v	s.e-z	Bemerkungen
	11111	00000	11111	
(P1: ANFO)	11111	11000	00111	
(P2: ANFO)	11111	12100	00111	y = (02100), P2 wird blockiert
(P3: ANFO)	11111	12210	00111	y = (00210), P3 wird blockiert, obwohl beide Gabeln frei sind - eine ist vorbestellt.
(P1: FREI)	22111	12210	10011	P2 wird deblockiert
----- U.S.W. -----				

4.3.2. Korrektheit und Verhungerungsfreiheit

Die folgenden Sätze zeigen die korrekte Funktion der Operationen ANFO/FREI-general bei der Verwendung zum Belegen und Freigeben von Betriebsmitteln, sowie die Vermeidung des Aushungerns.

Die Sätze gehen nicht von Prozessen aus, sondern von der möglichen Folge der Operationen.

Dazu formalisieren wir zunächst eine Berechnungsfolge:

Sei für $j=1,2,\dots,n$: $B(j) \in \mathbb{Z}^k$, $B(j) \geq 0$;

$B(j)$ wird als Bedarf der j -ten Anforderung angesehen.

Sei weiter $\alpha = a(1) a(2) \dots a(m)$

eine Folge von Operationen mit

$$a(t) = (op(t), id(t)),$$

$$op(t) \in \{ANFO, FREI\},$$

$$id(t) \in \{1, 2, \dots, n\}$$

und so daß

$$a) \quad (t_1 \neq t_2) \Rightarrow \\ (a(t_1) \neq a(t_2))$$

$$b) \quad (t_1 < t_2 \text{ and } op(t_1)=op(t_2)=ANFO) \Rightarrow \\ (id(t_1) < id(t_2))$$

$$c) \quad \text{für alle } j=1,\dots,n \text{ gibt es ein } t \text{ mit } a(t) = (ANFO, j)$$

Mit anderen Worten, keine Operation kommt zweimal vor; die Anforderungen stehen in der Reihenfolge

$$ANFO(B(1)), ANFO(B(2)), \dots, ANFO(B(n)),$$

dazwischen stehen die zugehörigen Freigaben in beliebiger Reihenfolge.

Seien für $t=0,1,\dots,m$ und für $j=1,\dots,n$ die Variablenwerte:

$$s(t).e, s(t).v, y(j) \in \mathbb{Z}^k,$$

(der erste Index t dient jetzt anders als bisher zur Unterscheidung der Werte zu verschiedenen Zeitpunkten; um die einzelnen Komponenten zu bezeichnen, schreiben wir bei Bedarf $s(t).e(i)$, $s(t).v(i)$).

Gelte weiter

- d) $s(0).v = 0, s(0).e \geq 0$
- e) $s(t).e = s(t-1).e', \text{ falls } op(t) = \text{ANFO},$
 $s(t).e = s(t-1).e + B(id(t)) \text{ sonst},$
- f) $s(t).v = s(t-1).v, \text{ falls } op(t) = \text{FREI},$
 $s(t).v = s(t-1).v + B(id(t)) \text{ sonst},$
- g) $y(j) = s(t).v(*) * \text{sgn}(B(j,*))$
 mit t so daß $a(t) = (\text{ANFO}, j)$, dies existiert nach c).
- h) definiere die binäre Funktion
 $\text{dblock}(j, t) := (s(t).e \geq y(j))$
- i) $op(t) = \text{FREI} \Rightarrow \text{dblock}(id(t), t-1)$

Die Bedingungen d) - g) besagen, daß die Variablen die durch die Operationen ANFO/FREI definierten Werte annehmen; die Funktion dblock gibt Auskunft, ob die j -te Anforderung zum Zeitpunkt t schon deblockiert ist; i) besagt, daß eine Freigabeoperation frühestens nach Deblockierung der zugehörigen Anforderung erfolgen darf.

Schließlich definiere die Summe der bis zu einem Zeitpunkt t zugeteilten Betriebsmittel:

$$j) \quad z(t) := \sum_{\substack{t' \leq t \text{ \& } op(t') = \text{ANFO} \\ \text{\& } \text{dblock}(id(t'), t)}} B(id(t'))$$

Damit sind die formellen Voraussetzungen für die beiden Sätze geschaffen.

Satz 1 (Korrektheit)

Es gilt für $t = 1, \dots, m$: $s(t).e - z(t) \geq 0$.

Das bedeutet nichts anderes, als daß keine nicht vorhandenen Betriebsmittel zugeteilt werden, genauer: daß ein Prozeß nur dann deblockiert wird, wenn sein Bedarf auch wirklich gedeckt ist.

Der Wert $s(t).e - z(t)$ entspricht den üblichen Semaphorwerten. Die Behauptung besagt also, daß diese nicht negativ werden.

Satz 2 (Aushungerungsfreiheit)

Angenommen, es gelten zusätzlich folgende Voraussetzungen:

1. für alle $j = 1 \dots n$ ist $B(j) \leq s(0).e$, d.h. keine Anforderung ist größer als der anfängliche Bestand an Betriebsmitteln.

2. für jedes j, t mit $\text{dblock}(j, t)$ gibt es ein t' mit

$$a(t') = (\text{FREI}, j),$$

d.h. jede deblockierte Anforderung führt auch irgendwann zur zugehörigen Freigabe.

Dann gilt:

für jedes $j = 1 \dots n$ gibt es ein t mit $\text{dblock}(j, t)$,

d.h. jede Anforderung wird irgendwann gewährt.

Beweis von Satz 1

Betrachte die i -te Komponente der Vektoren $s(t).e$ und $z(t)$; zu zeigen ist für jedes i : $s(t).e(i) \geq z(t, i)$.

Fall 1 : $z(t, i) = 0$

Hier ist die Ungleichung trivialerweise erfüllt, da $s.e$ nichtnegative Komponenten hat.

Fall 2 : $z(t, i) > 0$.

Betrachte dann den letzten positiven Summanden in der Summe

$$\begin{aligned}
 (1) \quad z(t,i) &= \sum_{\substack{t' \leq t \text{ \& } op(t')=ANFO \\ \& \text{ dblock}(id(t'),t)}} B(id(t'),i) \\
 &= B(id(t1),i) + B(id(t2),i) + \dots + B(id(tt),i);
 \end{aligned}$$

d.h. setze

$$\begin{aligned}
 (2) \quad tt &:= \max(t' : t' \leq t, op(t')=ANFO, \text{ dblock}(id(t'),t), B(t',i) > 0). \\
 \text{Ein solches gibt es wegen der Definition von } z(t) \text{ und wegen} \\
 z(t,i) &> 0.
 \end{aligned}$$

Anschaulich ist tt der Zeitpunkt der letzten Zuteilung für Betriebsmittel i .

Es ist also

$$(3) \quad B(id(t'),i) = 0 \quad \text{für} \quad tt < t' < t.$$

Folglich wegen (1) und (3):

$$\begin{aligned}
 (4) \quad z(t,i) &= \sum_{\substack{t' \leq tt \text{ \& } op(t')=ANFO \\ \& \text{ dblock}(id(t'),t)}} B(id(t'),i)
 \end{aligned}$$

Aus (2) folgt andererseits:

$$\text{dblock}(id(tt),t) \text{ und } B(id(tt),i) > 0,$$

daher nach Definition von dblock

$$(5) \quad s(t).e \geq y(id(tt)).$$

$$\begin{aligned}
 \text{Weiter ist } y(id(tt),i) &= s(tt).v(i) * \text{sgn}(B(id(tt),i)) \\
 &= s(tt).v(i).
 \end{aligned}$$

Also

$$(6) \quad s(t).e(i) \geq s(tt).v(i).$$

Nach Definition ist

$$\begin{aligned}
 (7) \quad s(tt).v(i) &= \sum_{t' \leq tt \text{ \& } op(t')=ANFO} B(id(t'),i)
 \end{aligned}$$

Nun ist sicher

$$(8) \quad \sum_{t' \leq tt \ \& \ op(t')=ANFO} B(id(t'),i) \qquad \sum_{\substack{t' \leq tt \ \& \ op(t')=ANFO \\ \& \ dblock(id(t'),t)}} B(id(t'),i)$$

da in der ersten Summe mehr Summanden vorkommen, und alle Summanden nicht-negativ sind.

Zusammenfassend folgt schließlich aus (6),(7),(8),(4):

$$s(t).e(i) \geq z(t,i)$$

Q.E.D.

Beweis von Satz 2

Der Beweis wird durch Induktion über j , den Index der Anforderungen, geführt.

Induktionsanfang : $j=1$

Für $j' < j$ gilt trivialerweise die Behauptung (de falso quodlibet).

Induktionsannahme:

Für alle $j' < j$ gelte die Behauptung:

es gibt t , so daß $dblock(j',t)$.

Zu zeigen ist dann:

es gibt t , so daß $dblock(j,t)$,

d.h. auch die j -te Anforderung wird deblockiert.

Sei für jedes $j'=1,2,\dots,j-1$ der Zeitpunkt $t1(j')$ so groß, daß gilt

$$(1) \quad dblock(j',t1(j'));$$

Dies existiert nach Induktionsannahme.

Sei weiter für jedes $j'=1,2,\dots,j-1$ der Zeitpunkt $t2(j')$ so, daß gilt

$$(2) \quad a(t2(j')) = (FREI, j').$$

Diesen gibt es nach Voraussetzung 2 des Satzes, und wegen (1).

Sei $tt2 := \max(t2(j'): j'=1,2,\dots,j-1)$.

$tt2$ ist also ein Zeitpunkt, zu dem alle Anforderungen mit $j' < j$ schon wieder freigegeben sind.

Es wird gezeigt, daß spätestens dann die j -te Anforderung auch deblockiert wird:

Es gilt nämlich

$$(3) \quad s(tt2).e \geq (s(0).e + \sum_{j' < j} B(j')),$$

da in $s(tt2).e$ wegen der Definition von $tt2$ mindestens die Summanden der rechten Seite eingehen.

Nach Voraussetzung 1 ist aber

$$(4) \quad s(0).e \geq B(j),$$

so daß nach (3) und (4):

$$(5) \quad s(tt2).e \geq \sum_{j' \leq j} B(j')$$

Die Summe auf der rechten Seite ist wiederum $= s(j).v$, und

$$(6) \quad s(j).v \geq y(j).$$

Somit ist $s(tt2).e \geq y(j)$, aber dies ist gerade die Deblockierungsbedingung für die j -te Anforderung:

$$dblock(j, tt2),$$

mit anderen Worten, die j -te Anforderung wird spätestens zum Zeitpunkt $tt2$ deblockiert,

Q.E.D.

Bemerkung:

Die spezielle Definition der Vektoren $y(j)$ mit der signum-Funktion geht nicht in den Beweis ein; Verhungerungsfreiheit wäre wegen (6) selbst dann noch gegeben, wenn man einfach $y := s.v$ setzte, also Blockierungen wegen fremder, nicht angeforderter Betriebsmittel nicht ausschloß.

Diskussion der Voraussetzungen von Satz 2:

Auf ein Prozeßsystem angewandt, besagen die Voraussetzungen:

1. Jede Anforderung ist, für sich allein genommen, ausführbar, d.h. übersteigt nicht den anfänglichen Bestand an Betriebsmitteln.
2. Die Operationen ANFO/FREI treten immer paarweise, in der richtigen Reihenfolge, und mit gleichen Parametern auf; bzw. jeder Prozeß gibt die belegten Betriebsmittel nach gewisser Zeit vollzählig zurück.

Für die Anwendung auf Belegungsprobleme sind diese Voraussetzungen gewiß realistisch und natürlich. Man könnte aber ANFO/FREI durchaus auch auf andere Probleme anwenden, wie die verwandten Operationen P/V. Für solche Anwendungen wäre dann Satz 2 nicht mehr richtig.

Die Notwendigkeit der ersten Voraussetzung ist unmittelbar einsichtig; dennoch muß darauf hingewiesen werden, daß bei ihrer Verletzung "alles schiefgeht": falls eine unmögliche Anforderung ($B(j) > s(0).e$) gestellt wird, bleibt nicht nur diese blockiert, sondern es kommt zu Verklemmungen auch anderer Anforderungen.

Im folgenden wird die Notwendigkeit von Voraussetzung 2 gezeigt.

Beispiel

Jeder der drei Prozesse "verbraucht" zwei Betriebsmittel und "erzeugt" zwei andere:

```
P: initially s.e = (1,1,0);
   process P1:  ANFO(1,1,0); ... FREI(0,1,1);
   process P2:  ANFO(0,1,1); ... FREI(1,0,1);
   process P3:  ANFO(1,0,1); ... FREI(1,1,0);
```

Dann ist folgender Ablauf möglich:

Aktion	s.e	s.v	s.e-z	Bemerkung
	(1,1,0)	(0,0,0)	(1,1,0)	
(P1:ANFO)	(1,1,0)	(1,1,0)	(0,0,0)	P1 zugeteilt
(P3:ANFO)	(1,1,0)	(2,1,1)	(0,0,0)	P3 blockiert, $y=(2,0,1)$
(P2:ANFO)	(1,1,0)	(2,2,2)	(0,0,0)	P2 blockiert, $y=(0,2,2)$
(P1:FREI)	(1,2,1)	(2,2,2)	(0,1,1)	P2 und P3 bleiben beide blockiert, da weder $(1,2,1) \geq (2,0,1)$ noch $(1,2,1) \geq (0,2,2)$.

Eigentlich müßte in dieser Situation P2 deblockiert werden: seine Anforderung (0,1,1) kann erfüllt werden. Das System P ist also nicht verklemmungsfrei. Beachte, daß das gleiche System bei Verwendung von P/V anstelle ANFO/FREI verklemmungsfrei ist.

(Ende des Beispiels).

Korollar zu Satz 2:

Unter den Voraussetzungen von Satz 2 gibt es insbesondere keine Verklemmung (deadlock).

denn jede Anforderung wird irgendwann deblockiert.

Deadlock, livelock und Verhungern

Das letzte Resultat soll Anlaß sein, diese Begriffe im Zusammenhang mit F-Systemen näher zu untersuchen.

Man sagt, ein Prozeß wird in einem Ablauf ausgehungert, wenn er nicht aktiv wird, obwohl er nicht dauernd blockiert ist. Dann gibt es zwei Fälle: entweder der Prozeß wird immer wieder aufgrund wechselnder Aktivitäten anderer Prozesse blockiert und deblockiert, oder er ist nie blockiert und wird trotzdem nicht aktiv. Den letzten Fall könnte man als grob unfaires Scheduling bezeichnen; er liegt außerhalb des Einflusses der eigentlichen Synchronisation.

Gewöhnlich meint man daher mit dem Phänomen des Aushungerns den ersten Fall:

ein Prozeß wird immer wieder blockiert und deblockiert und vom Scheduler nie aufgegriffen.

Im Fall des Philosophenproblems ist eine Form des Aushungerns bekannt, die wir hier zur Unterscheidung nach (ASHC75) "livelock" nennen wollen; sie wird etwa in (PIEP77) dargestellt und sieht in diesem Beispiel so aus:

ein Philosoph kommt nie zum Essen, weil immer einer seiner beiden Nachbarn ißt.

Allgemein: ein Prozeß bleibt immer blockiert, jedoch aufgrund dynamisch wechselnder Konstellationen; d.h. es wäre durchaus ein Ablauf möglich, der den Prozeß deblockieren würde.

Dagegen versteht man unter deadlock bekanntlich einen Zustand z, in welchem ein Prozeß notwendig für immer blockiert bleibt:

der Prozeß ist in jedem von z aus erreichbaren Zustand blockiert.

Auf dem Hintergrund dieser Begriffe ist nun unser Mechanismus wie folgt einzuordnen:

1. Verhungering im obigen Sinn kann bei F-Systemen prinzipiell nicht auftreten, da ein einmal deblockierter Prozeß nicht mehr blockiert werden kann, solange er nicht selbst aktiv wird.

2. Daß deadlock auch für ANFO/FREI-Systeme möglich ist, zeigt das obige Beispiel. Livelock dürfte für ANFO/FREI-Systeme nicht vorkommen, dies wurde jedoch nicht weiter untersucht und scheint auch von untergeordnetem Interesse angesichts der deadlock-Gefahr einerseits und von Satz 2 andererseits.

3. Satz 2 schließt für ANFO/FREI-Systeme der besonderen Form, die bei Belegungsproblemen entsteht, jede der drei diskutierten Arten von Verklemmung aus.

4. Für F-Systeme im allgemeinen kann weder deadlock noch livelock ausgeschlossen werden.

4.3.3. Grenzen der Anwendbarkeit

Die Sätze des letzten Abschnitts zeigen die Eignung des Operationen-paars ANFO/FREI für eine wichtige Klasse von Aufgabenstellungen: Belegungsprobleme sind damit ohne Gefahr von Aushungern oder livelock lösbar.

ANFO/FREI läßt sich als spezielle Form der P/V-general-Operationen auffassen, nämlich als eine sichere, aber nicht vollständige Realisierung der letzteren. Das heißt nichts anderes, als daß alle Abläufe, die ein bestimmtes ANFO/FREI-System zuläßt, auch im entsprechenden P/V-System aktiv sind, aber nicht umgekehrt. Das F-System schränkt also die Menge der aktiven Abläufe weiter ein - nämlich durch Realisierung einer "fairen" Strategie.

Nun wird diese Strategie in gewissen Fällen auch als unerwünschte Einschränkung empfunden werden. Ein solcher Fall ist Thema dieses Abschnitts. Er kommt zustande, indem Betriebsmittel ganz unterschiedlicher Charakteristik mit den gleichen Operationen koordiniert werden.

Beispiel

```
initially s.e=(1,1);
process P1:  ANFO(1,0); ... ; FREI(1,0);
process P2:  ANFO(1,1); ... ; FREI(1,1);
process P3:  ANFO(0,1); ... ; FREI(0,1);
```

Nehmen wir an, dabei stünde Betriebsmittel 1 für einen Drucker, Betriebsmittel 2 für einen Platten-Kanal.

Mit "unterschiedlicher Charakteristik" ist hier gemeint, daß der Drucker im allgemeinen für längere Zeit belegt wird - etwa in der Größenordnung von Minuten - , während der Platten-Kanal vielleicht nur für einige Transfers im Millisekundenbereich benötigt wird.

Betrachte dann folgenden Ablauf:

Aktion	s.e	s.v	s.e-z	
	(1,1)	(0,0)	(1,1)	
(P1:ANFO(1,0))	(1,1)	(1,0)	(0,1)	
(P2:ANFO(1,1))	(1,1)	(2,1)	(0,1)	P2 blockiert, y=(2,1)
(P3:ANFO(0,1))	(1,1)	(2,2)	(0,1)	P3 blockiert, y=(0,2)

Der Prozeß P3 wird hier blockiert, weil der von ihm benötigte Plattenkanal schon von P2 bestellt ist, und P2 noch auf den Drucker wartet. Im ungünstigsten Fall wird also P3 sehr lange blockiert, obwohl er nur für Millisekunden einen Kanal belegen wollte.

Man wird ein solches Verhalten also im allgemeinen nicht akzeptieren. Es erfüllt aber wohlgemerkt alle bisher formulierten Anforderungen wie Vermeidung von deadlock, livelock, Aushungern ! Man könnte dieses Verhalten geradezu als "übertriebene Fairness" bezeichnen.

Im Vergleich zum P/V-Mechanismus stellt man fest: Mit P/V-general-Operationen wäre der Prozeß P3 an der entsprechenden Stelle nicht blockiert, es käme also anscheinend nicht zu dem beobachteten Fehlverhalten. Tatsächlich wird dieses Verhalten - unnötig lange Verzögerung einer Operation - aber keinesfalls ausgeschlossen, sondern nur nicht erzwungen - weil eben P/V keine Strategieentscheidung enthält.

Mit Berücksichtigung der voraussichtlichen Belegungszeit könnte P3 natürlich ohne weiteres den Kanal zugeteilt bekommen, selbst ohne einen der anderen Prozesse zu behindern. Ein Koordinierungsmechanismus, der dies leisten sollte, müßte die Betriebsmittel in Klassen einteilen gemäß ihrer (durchschnittlichen) Belegungsdauer.

Außer einer Grenze für die sinnvolle Anwendbarkeit des ANFO/FREI-Mechanismus liefert dieses Beispiel noch eine wichtige Erkenntnis:

Die Anforderungen der Praxis werden durch keine der bisher formalisierten Eigenschaften eines Prozeßsystems vollständig wiedergegeben. Insbesondere sind in der Praxis Überlegungen über die zeitliche Dauer (etwa von Blockierungen) relevant, wo die Theorie nur Aussagen über die zeitliche Ordnung von Ereignissen erlaubt.

Schlichter ausgedrückt, es genügt eben oft nicht, zu wissen, daß ein Ereignis sicher irgendwann eintritt, sondern es macht einen Unterschied, ob dies in der nächsten Sekunde oder im nächsten Jahr geschieht.

4.3.4. Erweiterungen

Zu den P/V-Operationen gibt es eine Reihe von Verallgemeinerungen, von denen P/V-general keineswegs die umfassendste ist. Bathelt (BATH82) gibt eine ausführliche Darstellung dieser Vielfalt, wie etwa verschiedene Typen von Vektoradditions- und Vektorersetzungssystemen. Da die Operationen ANFO/FREI mit den bekannten Semaphor-Operationen verwandt sind, stellt sich die Frage, welche der höheren Mechanismen eine Entsprechung als F-Systeme haben.

Weit entfernt von einer vollständigen Antwort auf diese Frage, bringt dieser Abschnitt ein Beispiel einer Verallgemeinerung von ANFO/FREI. Seine V-System-Version geht auf Agerwala (AGER77) zurück und wird von Bathelt als pz/vz-System wie folgt beschrieben:

```

pz(S1,t1,d1; S2,t2,d2; ... ; Sn,tn,dn ) :=
    when (S1 ≥ t1 and ... and Sn ≥ tn )
    do  (S1 := S1-d1; ... ; Sn := Sn-dn );

vz(S1,d1; S2,d2; ... ; Sn,dn ) :=
    when true
    do  (S1 := S1+d1; ... ; Sn := Sn+dn );
  
```

mit $t_i, d_i \geq 0$ ($i=1, \dots, n$).

Wir wollen hier zusätzlich annehmen: $t_i \geq d_i$, damit die Semaphorwerte nicht negativ werden können.

Die Anforderung pz testet also die Semaphore mit den Vergleichswerten t_i , um dann die Differenzen d_i abzuziehen.

Für den Fall der Betriebsmittelbelegung läßt sich das so verstehen: Ein Prozeß belegt nur dann d_i Exemplare des i -ten Betriebsmittels, wenn mindestens t_i Exemplare frei sind. Die im letzten Abschnitt beobachteten Behinderungen lassen sich damit unter Umständen abmildern.

Um ein F-System nach dieser Idee zu definieren, schreiben wir die Parameter der Operationen wieder als Vektoren und lassen die Bezeichner S_i der Semaphore weg.

```
var s: array (1..n) of record e,v,b: integer end;
```

```
ANFOz ( t , d ) :=
  do   (s.b := max(s.b, s.v+t );
        s.v := s.v + d;
        y(*):= s.b(*) * sgn(t(*))
      )
  wait s.e > y;
```

```
FREIz ( d ) :=
  do   s.e := s.e + d;
```

Das Maximum ist wie üblich komponentenweise zu verstehen.

Die Variablen s.e und s.v spielen die bekannten Rollen als Zähler für Freigaben und Anforderungen. Neu kommt die Größe s.b hinzu, die jetzt benötigt wird, um die Fortsetzungsbedingung einer Anforderung von der Testgröße t abhängig zu machen. Für den Fall $t=d$ (bei allen Anforderungen) ergibt sich $s.b=s.v$, und alles stimmt mit den bekannten Operationen ANFO/FREI überein.

Die mit ANFOz/FREIz realisierte Strategie läßt sich so charakterisieren:

Eine Anforderung X wird dann deblockiert, wenn

- keine ältere Bestellung auf ein von X benötigtes Betriebsmittel wartet ,

und

- nach Abzug aller älteren Bestellungen der Bestand noch mindestens die Testgröße t erreicht. Bei den "älteren Bestellungen" zählen dabei immer nur die tatsächlich belegten Mengen d.

Diese beiden Bedingungen spiegeln die beiden Argumente der Maximum-Funktion in der Berechnung von s.b wider.

Ein Beispiel soll die Wirkung des Mechanismus verdeutlichen:

Der Übersichtlichkeit halber werden Vektoren ohne Klammern und Kommata geschrieben, also "210" statt "(2,1,0)".

Aktion	s.e	s.v	s.b	
	222	000	000	
(P1:ANFOz(210;110))	222	110	210	y=210, nicht blockiert
(P2:ANFOz(102;101))	222	211	212	y=202, nicht blockiert
(P3:ANFOz(012;011))	222	222	223	y=023, blockiert
(P4:ANFOz(001;001))	222	223	223	y=023, blockiert
(P2:FREIz(101))	323	223	223	es werden P3 und P4 gleichzeitig deblockiert
----- u.s.w. -----				

P3 zu deblockieren, ist hier korrekt, weil

(Bestand =) s.e-z = 112 $\geq$ 012 (= Testgröße t von P3).

P4 zu deblockieren, ist korrekt, weil selbst nach Zuteilung an P3:

(Bestand =) s.e-z = 101 $\geq$ 001 (= Testgröße t von P4).

Tatsächlich gelten auch hier die Analoga der Sätze 1 und 2 aus Abschnitt 4.3.1.; allerdings ist die Korrektheit etwas aufwendig zu formulieren; denn wenn wie im Beispiel mehrere wartende Anforderungen gleichzeitig deblockiert werden, braucht hinterher nicht die Bedingung "Bestand größer/gleich Testgröße" zu gelten.

Satz 1z:

Sei α eine Folge von Operationen ANFOz und FREIz, s.e die Anfangswerte der Semaphore. Ordne der Folge α eine Folge β von Operationen pz und vz derart zu, daß pz(t,d) immer an der Stelle steht, wo ANFOz(t,d) deblockiert wird; wenn mehrere gleichzeitig deblockiert werden, halte die zeitliche Reihenfolge der ANFOz-Operationen ein. Wähle die Anfangswerte der Semaphore entsprechend s.e.

Dann ist die entstehende Folge β zulässig im pz/vz-System.

Satz 2z: (Aushungerungsfreiheit)

Sei α eine Folge von Operationen $ANF0z$ und $FREIz$ mit folgenden zusätzlichen Eigenschaften:

1. Für keine Anforderung $ANF0z(t,d)$ ist die Testgröße t größer als der Anfangswert s.e der Semaphore.
2. Zu jeder Anforderung $ANF0z(t,d)$, die deblockiert wird, gibt es eine zugehörige Freigabe mit der gleichen Größe d : $FREIz(d)$.

Dann wird jede Anforderung in α irgendwann deblockiert.

Die Beweise sind denen aus Abschnitt 4.3.2 ähnlich.

4.4. Das zweite Leser-Schreiber-Problem

Dieses Problem wurde in Abschnitt 2.3. vorgestellt. Das Charakteristische war die Priorität der Schreiber vor den Lesern. Hier stellt sich wie beim 2-Erzeuger-2-Verbraucher-Problem die Frage nach dem genauen Verständnis von "Priorität"; in der ursprünglichen Aufgabenstellung war gefordert worden: "sobald ein Schreibaufwurf ansteht, darf kein weiterer Leser zugelassen werden". Damit kann man natürlich ein Aushungern (genauer: livelock) der Leser nicht ausschließen. Wie auch beim 2-Erzeuger-2-Verbraucher-Problem läßt sich leicht eine Lösung als F-System vom Zähler-Typ finden, die der Fairness-Anforderung genügt.

In Anlehnung an den "bolt"-Mechanismus der Sprache PEARL (KAMR79) wollen wir die verwendeten Synchronisationsoperationen

enter/leave für nicht-exklusiven Zugriff (Leser) und
reserve/free für exklusiven Zugriff (Schreiber)

nennen.

Sie seien wie folgt definiert:

```

var s: record e,v, xe,xv: integer end;

enter(s) :=
  do s.v := s.v+1; y := s.xv wait s.xe ≥ y;

leave(s) :=
  do s.e := s.e+1;

reserve(s) :=
  do y := (s.v, s.xv); s.xv := s.xv+1 wait (s.e, s.xe) ≥ y;

free(s) :=
  do s.xe := s.xe+1;
  
```

Damit läßt sich die Aufgabe wie folgt lösen:

```

initially s.e = s.v = s.xe = s.xv = 0;
process READER: loop enter(s); read; leave(s) endloop;
process WRITER: loop reserve(s); write; free(s) endloop;
  
```

Begründung:

Die Fortsetzungsbedingungen haben folgende Bedeutung:

- bei enter: alle exklusiven Zugriffe (reserve, WRITER), die zum Zeitpunkt des Aufrufs von enter schon begonnen waren - deren Zahl gibt y an - , sind beendet.
- bei reserve:
 - $s.e \geq y(1)$ - alle nicht-exklusiven Zugriffe (enter, READER), die zum Zeitpunkt des Aufrufs von reserve schon begonnen waren, sind beendet, und
 - $s.xe \geq y(2)$ - alle exklusiven Zugriffe (reserve von anderen WRITER-Prozessen), die zum Zeitpunkt des Aufrufs von reserve schon begonnen waren, sind beendet.

Insbesondere ergeben sich folgende Eigenschaften:

Solange noch kein Schreiber aktiv war, können die Leser unbehindert (ohne Blockierung) zugreifen.

Wenn sich ein Schreiber anmeldet (reserve), wird kein weiterer Leser zugelassen, und der Schreiber bleibt solange blockiert - wegen der ersten Komponente seiner Fortsetzungsbedingung - , bis die noch tätigen Leser ihre Tätigkeit beendet haben.

Ein zweiter Schreiber blockiert sich wegen der zweiten Komponente seiner Fortsetzungsbedingung, bis der erste fertig ist.

Ein in der Zwischenzeit "ankommender" Leser wird zwar blockiert, aber nur solange, bis alle vor ihm angemeldeten Schreiber fertig sind.

Als entscheidender Punkt für die "Fairness" läßt sich also festhalten - und das gilt für alle hier vorgestellten Mechanismen - :

Keine Operation op1 kann behindert werden durch eine andere Operation op2, die sich später als op1 angemeldet hat.

Man könnte das eine schwache Form des FIFO-Prinzips nennen. Es läßt sich etwas strenger so ausdrücken:

Sei

$$\alpha = \alpha_1 \text{ op1 } \alpha_2 \alpha_3$$

ein aktiver Ablauf und op1 nach α nicht blockiert.

Sei op2 eine beliebige Operation, dann ist op1 nach

$$\alpha_1 \text{ op1 } \alpha_2 \text{ op2 } \alpha_3$$

nicht blockiert.

Dabei heißt eine Operation blockiert (war bisher nur für Prozesse definiert), wenn seit ihrem Aufruf ihre Fortsetzungsbedingung noch nie erfüllt war.

Daß dieses Prinzip keine wirkliche FIFO-Bearbeitung der Anforderungen bedeutet, ist aus den bisherigen Beispielen schon klar geworden; die Fortsetzung einer Operation kann sich ja durchaus aufgrund älterer Anforderungen beliebig verzögern.

Anmerkungen

1. Die hier vorgeschlagene Strategie ist nicht völlig neu. Sie gleicht vielmehr weitgehend der von Hoare (HOAR74) formulierten "Strategie der alternierenden Prioritäten", die in der Literatur zum Leser-Schreiber-Problem seitdem mehrmals (LAVE78, HOFM82, MACK83) aufgegriffen wurde. Hier liegt der Schwerpunkt auf der, wie ich meine, einfachen Darstellungsmöglichkeit als F-System.

Dennoch sei der feine Unterschied zu Hoare's Lösung herausgestellt: Er "schaltet" nämlich beim Ende eines jeden Schreibzugriffs die Prioritäten "um" und läßt alle inzwischen wartenden Leser zur Ausführung zu. Er würde also nach dem Ablauf

(WRIT1:reserve) (WRIT2:reserve) (READ1:enter) (WRIT1:free)

als nächstes den wartenden Leser READ1 fortsetzen (= zulassen). Mein Algorithmus bevorzugt in diesem Fall den (älteren) Schreiber WRIT2.

2. Eingangs wurde dargelegt, wie die unterschiedliche Auffassung des Begriffs Priorität zu den verschiedenen Lösungen des Leser-Schreiber-Problems führt. Mackert (MACK83) hat erstmals die Mehrdeutigkeit dieses Begriffs erkannt und zwei Präzisierungen des Prioritätsbegriffs gegeben, sog. schwache und starke Priorität. Keine davon stimmt allerdings mit der hier spezifizierten Strategie der "fairen Priorität" überein.

3. Laventhal (LAVE78) gibt eine Implementierung der Strategie der alternierenden Prioritäten an, die auf Pfadausdrücken basiert und Zähler für die Operationen read-request, write-exit etc. verwendet. Die Verwendung dieser Zähler ähnelt unserem Konzept der F-Systeme vom Zähler-typ. Im Vergleich zu diesen erlauben Pfadausdrücke eine flexiblere Beschreibung von parallelem Verhalten, sind aber auch komplizierter und schwerer verständlich. In einem von (MACK83) analysierten Beispiel sind die Pfadausdrücke so kompliziert, daß Fehler unvermeidlich erscheinen.

4.5. Two-way-bounded-Mechanismen

Alle bisher definierten Mechanismen verwenden integers als Datenbasis und sind somit potentiell unendlich. Da digital realisierte Systeme immer nur endlich viele Zustände kennen, ist die Frage von Interesse, welche Synchronisationsmechanismen mit nur endlich vielen Zuständen auskommen. In Abschnitt 2. wurde schon erwähnt, daß die Siemens-Koordinierungszähler mit Unter- und Obergrenze arbeiten, deren Rollen zueinander symmetrisch sind.

Das entsprechende V-System ist von Lipton (LIPT74) als P/V-two-way-bounded-System bekannt und so definiert:

Sei k eine natürliche Zahl.

$$\begin{aligned} Pk(s) &:= \text{when } s > 0 \text{ do } s := s-1; \\ Vk(s) &:= \text{when } s < k \text{ do } s := s+1; \end{aligned}$$

wobei $0 \leq s \leq k$.

Aufgrund der Symmetrie können hier nicht nur P-, sondern auch V-Operationen zu Blockaden führen. Der Mechanismus bietet sich z.B. an zur Koordinierung eines Puffers der Größe k : bei leerem Puffer blockiert sich der Verbraucher, bei vollem der Erzeuger.

Da der Semaphor nur endlich viele Werte annehmen kann, kommt dieses System mit endlich vielen Zuständen aus. Das gilt nicht ohne weiteres für die entsprechende F-System-Version:

$$\begin{aligned} ANFOk(s) &:= \text{do } (s.v := s.v+1; y := s.v) \\ &\quad \text{wait } s.e \geq y; \\ FREIk(s) &:= \text{do } (s.e := s.e+1; y := s.e-k) \\ &\quad \text{wait } s.v \geq y; \end{aligned}$$

Hier gilt

$(0 \leq (s.e-s.v) \leq k)$ g.d.w. keine Operation ist blockiert.

$0 > (s.e-s.v)$ g.d.w.

mindestens eine ANFO-Operation ist blockiert.

$(s.e-s.v) > k$ g.d.w.

mindestens eine FREI-Operation ist blockiert.

Wegen der unendlichen Wertebereiche der Variablen hat dieser Mechanismus nicht mehr die oben dargelegte Eigenschaft der Endlichkeit. Da nicht die Absolutwerte der Zähler $s.e$ und $s.v$, sondern nur deren Differenz von Interesse ist, könnte man versuchen, durch eine Art Überlaufbehandlung doch noch einen endlichen Zustandsraum zu erreichen. Das scheitert jedoch im allgemeinen an der Grundeigenschaft von F-Systemen, daß jeder Aufruf einen Zustandübergang bewirkt: nimmt man die Anzahl der Prozesse als nicht bekannt an, dann können beliebig viele Anforderungen nacheinander eintreffen, und die Werte $s.v$ und $s.e$ laufen beliebig weit auseinander.

Bei beschränkter Prozeß-Anzahl führt eine solche Überlauf-Behandlung tatsächlich zu einem endlichen Zustandsraum. Sie wird in Abschnitt 4.7. für alle Systeme vom Zähler-Typ untersucht.

Die hier vorgestellten Operationen arbeiten nur mit einem einzelnen Semaphor und Inkrement 1 - eine chunk- oder multiple-Version der zweiseitig begrenzten Operationen erscheint nicht sinnvoll; für eine solche würde nämlich nicht mehr gelten, daß zu einem Zeitpunkt nur entweder ANFO- oder FREI- Operationen blockiert sind.

4.6. Die Leistungshierarchie

In diesem Abschnitt werden die bisher definierten Mechanismen miteinander verglichen. Als Grundlage des Vergleichs benutzen wir die Klasse SNZ der "schwach nicht-zählenden" Schedules. Für die Simulation bezüglich dieser Klasse gilt nach Abschnitt 3.6. ein Invarianztheorem; dieses erlaubt einen Nachweis von Unterschieden durch lokalen Vergleich.

Definition:

Seien A und B Typen von F-Systemen. (Ein solcher Typ oder Mechanismus ist gegeben durch Angabe der synchronisierenden Aktionen).

1. Dann sagen wir, der Mechanismus A umfaßt den Mechanismus B, wenn gilt: Jedes Prozeßsystem vom Typ B kann durch ein Prozeßsystem vom Typ A bezüglich der Scheduleklasse SNZ simuliert werden.

2. Weiter heißt der Mechanismus A stärker als der Mechanismus B, wenn außerdem gilt:

Es gibt ein Prozeßsystem vom Typ A, welches durch kein Prozeßsystem vom Typ B bezüglich SNZ simuliert werden kann.

(Mit anderen Worten: A umfaßt B und B umfaßt nicht A).

Dieses Vergleichskonzept entspricht genau dem von Lipton, angewandt auf unseren Simulationsbegriff.

Abbildung 1 zeigt die meisten der bisher definierten Mechanismen, mit den Beziehungen, die schon aufgrund der Definitionen gelten: in dieser Abbildung führt ein gerichteter Pfad von A nach B genau dann, wenn jeder Mechanismus vom Typ B per definitionem auch zum Typ A gehört. Dann umfaßt A selbstverständlich B, aber es ist noch nicht bekannt, ob "A stärker B" gilt.

Um solche Aussagen machen zu können, benutzen wir den lokalen Vergleich, und definieren:

Definitions-Hierarchie

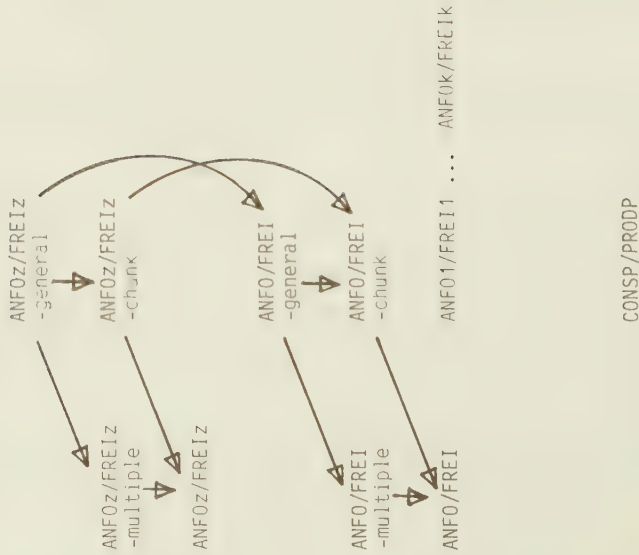
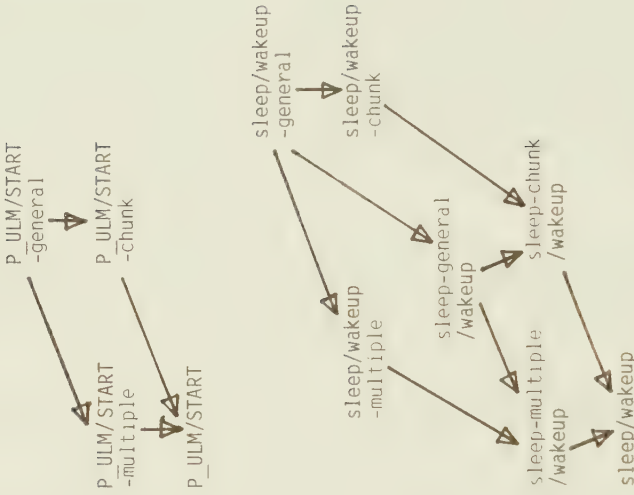


Abbildung 1



Lokale Leistungshierarchie

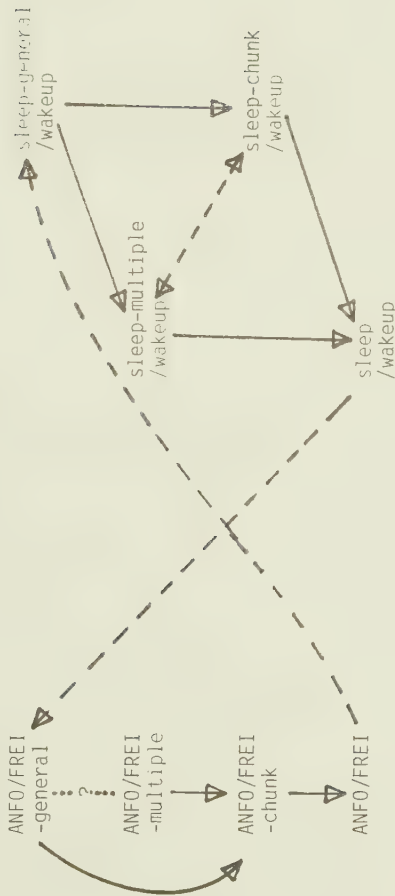


Abbildung 2

Anmerkung: die Bezeichnungen sleep-general/wakeup etc. bezeichnen die Varianten mit allgemeinem "sleep", aber nur einfachem "wakeup".

1. Ein Mechanismus A umfaßt lokal den Mechanismus B, wenn jeder Schnitt, der von einem Prozeßsystem vom Typ B definiert wird, auch von einem Prozeßsystem vom Typ A definiert wird.

2. Ein Mechanismus A ist lokal stärker als der Mechanismus B, wenn es außerdem ein F-System P vom Typ A und einen Schnitt X gibt, so daß P den Schnitt X definiert; und kein F-System vom Typ B definiert den Schnitt X.

Folgende einfache Tatsachen folgen direkt aus den Definitionen:

Lemma:

1. Wenn (A stärker B), dann (A umfaßt B)
2. Wenn (A lokal stärker B), dann (A umfaßt lokal B)
3. Wenn (A umfaßt B), dann (A umfaßt lokal B)
4. Wenn (A umfaßt B) und (A lokal stärker B), dann (A stärker B).
5. Alle vier Relationen sind transitiv.

Im Rest des Abschnitts beweisen wir ein

Lokales Leistungshierarchie-Theorem:

Zwei in der Abbildung 2 enthaltene Mechanismen A und B stehen dann in der Relation "A lokal stärker als B", wenn ein durchgezogener Pfeil oder ein gerichteter Pfad aus durchgezogenen Pfeilen von A nach B eingezeichnet ist.

Falls ein gestrichelter Pfeil von A nach B eingezeichnet ist, gilt: B umfaßt nicht lokal A.

Beweis:

Es ist zu zeigen: ANFO/FREI-mult umfaßt ANFO/FREI-chunk lokal.

Alle anderen behaupteten "umfaßt"-Relationen ergeben sich schon aus Abbildung 1.

Es bleiben also nur die Unterschiede nachzuweisen:

Für je zwei Mechanismen A und B, für die ein Pfeil - durchgezogen oder gestrichelt - von A nach B eingezeichnet ist, ist ein Schnitt zu finden, der durch A, aber nicht durch B definiert wird.

Diese Nachweise werden in den nächsten Abschnitten erbracht.

Anmerkungen

1. Die Bezeichnungen ANF01/FREI1 ... ANFOk/FREIk stehen für die two-way-bounded-Systeme.
2. Der Mechanismus PROD1/PROD2/CONS1/CONS2, der für das 2-Erzeuger-2-Verbraucher-Problem eingeführt wurde, ist hier wegen seiner begrenzten Anwendbarkeit nicht aufgenommen, ebenso der "bolt"-Mechanismus enter/leave/reserve/free aus Abschnitt 4.4.
3. Die in 4.1. als PROD/CONS definierten Operationen sind offensichtlich identisch mit dem einfachen ANFO/FREI, ebenso die Operationen KOOR+/- aus 2.1.
4. Wegen der Transitivität der Relationen folgt aus dem Theorem die Unvergleichbarkeit zweier Mechanismen A und B, falls je ein gerichteter Pfad von A nach B und von B nach A eingezeichnet sind, z. B. für die Systeme ANFO/FREI-chunk und sleep/wakeup.
5. Die Hierarchie ist noch nicht vollständig ! Zm einen wurden bei weitem nicht alle definierten Mechanismen eingeordnet, zum anderen ist die Frage noch offen, ob ANFO/FREI-general lokal stärker ist als ANFO/FREI-multiple oder nicht.
6. Von den verschiedenen sleep/wakeup-Varianten wurden nur die eingeordnet, bei denen "wakeup" nur ein einzelnes Ereignis signalisiert, d.h. die Verallgemeinerung bezieht sich nur auf "sleep".
7. Die folgenden Unterabschnitte beziehen sich ausschließlich auf lokalen Vergleich, daher wird das Wort "lokal" gelegentlich weggelassen.

4.6.1. ANFO/FREI-chunk stärker ANFO/FREI

Das ANFO/FREI-chunk-Prozeßsystem

```
initially s.e = 3;
process P1:ANFO(1); process P2:ANFO(2); process P3:ANFO(2);
```

definiert folgenden Schnitt:

```
X(123) = {12}, X(132) = {13}, X(213) = {12},
X(231) = {2}, X(312) = {13}, X(321) = {3}.
```

(Wo es ohne Mißverständnis möglich ist, schreiben wir (123) für die Folge (1,2,3) und {12} für die Menge 1,2 usw.).

Es sei an die Bedeutung des Schnitts erinnert: Nach dem Ablauf

```
(P1:ANFO(1)) (P2:ANFO(2)) (P3:ANFO(2))
```

sind genau die Prozesse P1 und P2 deblockiert, usw.

Es gilt nun zu zeigen, daß der Schnitt X von keinem (einfachen) ANFO/FREI-System Q definiert wird.

Angenommen, das sei der Fall.

Beobachtung: Im Schnitt X ist jeder der drei Prozesse bei gewissen Abläufen blockiert, bei anderen Abläufen nicht.

Die drei Aktionen in Q, die zur Definition des Schnitts dienen, müßten daher Anforderungen für den gleichen Semaphor sein.

Wegen $X(123) = \{12\}$ mußte der Anfangswert dieses Semaphors 2 sein. Dann würde sich aber $X(231) = \{23\}$ ergeben. WIDERSPRUCH.

4.6.2. ANFO/FREI-mult stärker als ANFO/FREI-chunk

Wir haben zunächst zu zeigen: ANFO/FREI-mult umfaßt lokal ANFO/FREI-chunk, d.h. sei X ein Schnitt, P ein ANFO/FREI-chunk-Prozeßsystem, und P definiere X; dann ist ein ANFO/FREI-mult-Prozeßsystem Q zu konstruieren, das X definiert.

Man darf annehmen:

1. In P hat jeder Prozeß nur eine einzige Aktion - mehr trägt er zur Definition eines Schnitts nicht bei.
2. Diese Aktionen sind synchronisierende Aktionen.
3. Sie beziehen sich alle auf den gleichen Semaphor - sonst wird die Konstruktion einfach für jeden Semaphor durchgeführt; die entstehenden Teilsysteme beeinflussen einander nicht.
4. Alle Aktionen in P sind Anforderungen: da für die Definition eines Schnitts nur der Endzustand nach jeder Ausführungsreihenfolge eine Rolle spielt, kann man FREI-chunk-Operationen eliminieren, indem man den Anfangswert des Semaphors entsprechend erhöht.

P besteht also aus n Operationen

ANFO(B1), ANFO(B2), ... , ANFO(Bn);

der Anfangswert sei A, wobei $B_1, \dots, B_n, A$ Ganzzahlen sind.

Wir konstruieren Q mit 2^n Semaphoren, die mit den Teilmengen von $\{1, 2, \dots, n\}$ indiziert sind:

$S(\emptyset), S(\{1\}), S(\{2\}), \dots, S(\{1, 2, \dots, n\})$.

Der Prozeß Q_i fordert eine Menge M_i von Semaphoren an, die folgendermaßen definiert wird:

für $U \subseteq \{1, \dots, n\}$ gilt: $S(U) \in M_i \iff i \in U$.

Soweit hängt die Konstruktion nur von n, nicht von P ab. Die das Prozeßsystem P definierenden Größen $B_1, \dots, B_n, A$ gehen in die Konstruktion nur über die Anfangswerte der 2^n Semaphore ein:

Sei nämlich der Anfangswert des Semaphores $S(U)$:

$$\begin{aligned} A(U) &= \text{card}(U), \quad \text{falls} \quad \sum_{i \in U} B(i) > A, \\ &= \text{card}(U) - 1 \text{ sonst.} \end{aligned}$$

(card(U) ist die Kardinalität der Menge U).

Der erste Fall bedeutet, daß sich an diesem Semaphor keiner der Prozesse in Q je blockieren kann - man kann diesen Semaphor auch weglassen. Im zweiten Fall wird die letzte Operation, die diesen Semaphor anfordert, blockiert.

Nun ist zu zeigen, daß Q den Schnitt X definiert; da die Konstruktion symmetrisch ist bezüglich einer Permutation der Zahlen 1..n, genügt es den Ablauf $Q_1 Q_2 \dots Q_n$ zu betrachten, mit $Q_i = \text{ANFO}(M_i)$. Er muß die

gleichen Prozesse blockiert hinterlassen wie der Ablauf
 $\text{ANFO}(B_1) \text{ ANFO}(B_2) \dots \text{ANFO}(B_n)$.

1. Angenommen P_i ist blockiert, dann ist $B_1+B_2+\dots+B_i > A$,
 daher für $U = \{1,2,\dots,i\}$: $A(U) = i-1$;
 nun haben aber alle Aktionen $Q_1, Q_2, \dots, Q_i$ den Semaphor $S(U)$
 angefordert. Daher ist Q_i blockiert.

2. Angenommen Q_i ist blockiert, dann ist
 entweder schon ein früheres Q_j ($j < i$) blockiert; in diesem Fall darf
 man per Induktion annehmen, daß auch P_j blockiert ist, dann aber auch
 P_i , weil P_j den einzigen Semaphor für sich "bestellt" hat;
 oder es gilt für eine Unterfolge $(t_1, t_2, \dots, t_t = i)$ von $(1, 2, \dots, i)$:
 $B(t_1) + B(t_2) + \dots + B(t_t) > A$;
 auch in diesem Fall ist P_i blockiert.

z.E.D.

Bemerkung:

Ein entsprechendes Resultat wurde von Dolev (DOLE77,79) für die Systeme
 P/V -general und P/V -multiple bewiesen. Die dortige Konstruktion
 funktioniert ähnlich der obigen.

Möglicherweise läßt sich der obige Beweis auch auf ANFO/FREI -multiple
 übertragen.

Jetzt ist noch ein Gegenbeispiel für die umgekehrte Relation anzugeben:
 Das ANFO/FREI -multiple-Prozeßsystem

```
initially s.e = (1,1,0);
process P1:ANFO(1,0,0);
process P2:ANFO(1,1,0);
process P3:ANFO(0,1,1);
```

definiert folgenden Schnitt:

```
x(1,2,3) = 1,    x(1,3,1) = 1,    x(2,1,2) = 1,
x(2,2,1) = 1,    x(1,1,1) = 1,    x(2,2,1) = 0
```

Es ist nun zu zeigen, daß kein ANFO/FREI-chunk-System diesen Schnitt definiert.

Ein solches müßte aus Anforderungen

$$\text{ANFO}(a), \text{ANFO}(b), \text{ANFO}(c)$$

an den gleichen Semaphor bestehen, mit $a, b, c \geq 0$, und mit Anfangswert $s.e = S$.

Aus den in X gegebenen Blockierungen folgt:

$$c > S \quad \text{wegen} \quad (3 \notin X(312)),$$

$$c+a \leq S \quad \text{wegen} \quad (1 \in X(312)).$$

WIDERSPRUCH.

4.6.3. sleep-chunk/wakeup und sleep-mult/wakeup sind unvergleichbar

Um zu zeigen:

sleep-general/wakeup	stärker	sleep-mult/wakeup,
sleep-general/wakeup	stärker	sleep-chunk/wakeup,
sleep-mult/wakeup	stärker	sleep/wakeup,
sleep-chunk/wakeup	stärker	sleep/wakeup,

genügt es zu zeigen, daß die chunk- und die mult-Version miteinander unvergleichbar sind. Dies geschieht durch die Angabe zweier Beispiele:

Das sleep/wakeup-multiple-Prozeßsystem

```
process P1:sleep(s1,s2);
process P2:wakeup(s1);
process P3:wakeup(s2);
process P4:wakeup(s2);
```

definiert den Schnitt X , auf dessen vollständige Angabe wir verzichten.

Es ist nun zu zeigen, daß kein sleep-chunk/wakeup-System diesen Schnitt definiert.

Angenommen, dies geschehe mit Operationen Q_1, Q_2, Q_3, Q_4 .

Dann müssen wegen

$$X(3124) = \{1234\}, \quad X(3214) = X(3412) = \{234\}$$

die Operationen von der Form sein

$$Q_1 = \text{sleep}(s, 2), \quad Q_2 = \text{wakeup}(s), \quad Q_4 = \text{wakeup}(s),$$

(denn Q1 wird von nachfolgendem Q2 und Q4 deblockiert, nicht aber von einem der beiden allein).

Es sei daran erinnert, daß die wakeup-Operation nach unserer Definition nur $s.v := s.v + 1$ bewirkt - "chunk" bezieht sich nur auf sleep.

Weiter folgt wegen

$$X(4123) = \{1234\},$$

daß $Q3 = \text{wakeup}(s)$.

Dann ist aber Q1 nach dem Ablauf Q2 Q1 Q3 Q4 ebenfalls nicht blockiert, im Widerspruch zu $X(2134) = \{234\}$.

Für die umgekehrte Richtung betrachten wir das sleep-chunk/wakeup-Prozeßsystem

```
process P1:sleep(s,2);
process P2:wakeup(s,1);
process P3:wakeup(s,1);
process P4:wakeup(s,1);
```

Dies definiert einen Schnitt X.

Angenommen, ein sleep/wakeup-mult-System definiere X mit den Operationen Q1,Q2,Q3,Q4.

Wegen $X(3214) = \{234\}$ und $X(3124) = \{1234\}$

mußte dann Q1 ein Ereignis erwarten, welches von Q2 signalisiert wird:

Beachte dabei, daß wakeup sich nur auf ein einziges Ereignis beziehen kann :

$$Q1 = \text{sleep}(s2, \dots ? \dots); \quad Q2 = \text{wakeup}(s2).$$

Aus Symmetriegründen gilt aber die gleiche Überlegung für Q3 und Q4:

$$Q1 = \text{sleep}(s2, s3, s4); \\ Q2 = \text{wakeup}(s2); \quad Q3 = \text{wakeup}(s3); \quad Q4 = \text{wakeup}(s4);$$

Aber dieses System definiert nicht den Schnitt X, weil z.B.

$$X(2134) = \{1234\}, \text{ aber}$$

nach (Q2 Q1 Q3 Q4) ist Q1 blockiert.

Damit ist die Teil-Hierarchie der vier sleep/wakeup-Mechanismen gezeigt.

4.6.4. ANFO/FREI und sleep/wakeup sind unvergleichbar

Um die Unvergleichbarkeit jeder Version des ANFO/FREI-Mechanismus mit jeder Version von sleep/wakeup zu zeigen, genügt es, den höchsten Mechanismus jeder Gruppe mit dem niedrigsten der anderen zu vergleichen. Wir zeigen also anhand zweier Beispiele:

- a) Es gibt einen Schnitt, der von sleep/wakeup definiert wird, jedoch nicht von ANFO/FREI-general.
- b) Es gibt einen Schnitt, der von ANFO/FREI definiert wird, jedoch nicht von sleep/wakeup-general.

Zu a): Hier genügt das einfachste denkbare Beispiel, das Prozeßsystem

```
process P1: sleep(s);
process P2: wakeup(s);
```

Es definiert den Schnitt : $X(12) = \{12\}$, $X(21) = \{2\}$.

Angenommen, dieser Schnitt würde von zwei Operationen Q1, Q2 des Typs ANFO/FREI-general definiert; dann kann Q1 nur eine Anforderung sein, da es in einem Fall blockiert wird.

Wäre Q2 eine Freigabe, dann müßte $X(12)=X(21)$ sein, weil für die Fortsetzungsbedingung $s.e \geq y$ die Reihenfolge der Operationen keine Rolle spielt - in die Größe y geht (anders als bei sleep/wakeup) der Wert s.e nicht ein. (Allgemein kann man Operationen FREI immer nach links verschieben, ohne das Gesamtergebnis zu ändern; das ist die aus (LIPT75) bekannte Tatsache, daß V-Operationen "left-mover" sind).

Wir haben also

$$Q1 = \text{ANFO}(d1); \quad Q2 = \text{ANFO}(d2);$$

mit Vektoren d1,d2.

Wir nehmen A als Anfangswert des Zählervektors an.

Dann muß, um X zu definieren, gelten:

$$\begin{aligned} A &\geq (d1+d2)*\text{sgn}(d2), \\ \text{not } (A &\geq (d2+d1)*\text{sgn}(d1)); \end{aligned}$$

Die letzte Ungleichung bedeutet aber, daß für eine Komponente j:

$$A(j) < d2(j) + d1(j), \quad d1(j) > 0.$$

Wegen $A \geq d1$ muß aber auch $d2(j) > 0$ sein, also

$$A(j) < (d1(j) + d2(j)) * \text{sgn}(d2(j)), \quad \text{WIDERSPRUCH.}$$

Anmerkung:

letzten Endes wurde damit gezeigt: wenn von mehreren Anforderungen, in einer bestimmten Reihenfolge gestellt, keine blockiert wird, dann gilt das auch bei jeder anderen Reihenfolge.

Dies gilt für ANFO-general, jedoch nicht für die erweiterten Operationen ANFOz.

Zu b): auch hier genügt glücklicherweise ein äußerst simples Beispiel:

Das Prozeßsystem

initially s.e=1;

process P1: ANFO(s); process P2: ANFO(s);

definiert den Schnitt $X(12) = \{1\}$, $X(21) = \{2\}$.

Angenommen Q1 und Q2 vom Typ sleep/wakeup-general definierten X.

Da sich beide Operationen blockieren können, müssen beide vom Typ "sleep(...)" sein. Dann aber sind beide in jedem Fall blockiert:

$$X(12) = X(21) = 0.$$

WIDERSPRUCH.

Schlußbemerkung zur lokalen Leistungshierarchie:

Die Ergebnisse überraschen kaum. Die Mechanismen ANFO/FREI und sleep/wakeup sind schon von den Definitionen her so unterschiedlich (nämlich gepuffert bzw. ungepuffert, vgl. 4.2.5.), daß man nachweisbare Unterschiede erwarten kann. Entsprechend leicht fällt auch der Beweis des letzten Abschnitts. Weniger offensichtlich ist die in 4.6.2. nachgewiesene Beziehung zwischen ANFO/FREI-multiple und -chunk, deren Definitionen ja zunächst Unvergleichbarkeit erwarten lassen.

4.7. Zur Implementierung

Die F-Systeme vom Zählertyp wurden nicht zuletzt deswegen näher untersucht, weil sie relativ leicht und effizient zu implementieren sind. Jede Implementierung eines Synchronisationsmechanismus muß ja die Frage beantworten, wie für jede wartende (blockierte) Operation die Fortsetzungsbedingung dargestellt wird; nach dem Aufruf einer solchen Operation ist diese Bedingung dem Betriebssystem bekannt, also nicht nur implizit im Code des Prozesses selbst enthalten. Die Darstellung sollte die Auswertung der Bedingungen erleichtern, damit der Mechanismus bei jeder Aktion, die möglicherweise andere deblockiert, erkennt, welche Prozesse zu deblockieren sind.

Die spezielle Form der Systeme vom Zähler-Typ unterstützt nun die Handhabung dieser Bedingungen, denn es genügt, für jede Aktion den Vektor y zu speichern, da alle Fortsetzungsbedingungen die Form haben $s \geq y$.

Am Beispiel von ANFO/FREI soll hier die Implementierung erläutert werden. Dabei ist noch zu beachten, daß die Zähler in der Theorie über alle Schranken wachsen können, was in der Praxis zwangsläufig zu Überläufen führen würde. Diese lassen sich vermeiden, solange die zueinander dualen Zählervektoren $s.e$ und $s.v$ nicht zu weit "auseinanderlaufen": wenn allerdings für eine Komponente j

$$\text{abs}(s.e(j) - s.v(j)) > \text{maxint},$$

wo maxint die größte darstellbare Ganzzahl ist, dann versagt die Implementierung; dies tritt aber nur auf, wenn entweder Anforderungen oder Freigaben um die betreffende große Zahl voreilen: diese Einschränkung stört in der Praxis nicht, und haftet im übrigen jeder Implementierung an, nicht nur den Systemen vom Zählertyp.

Für das Beispiel brauchen wir folgende Voraussetzungen:

- Es soll ein Datentyp `list of ...` zur Verfügung stehen mit den Operationen `insert`, `remove`, der Konstanten `emptylist`, und mit der Sprachkonstruktion

```
for (..Element..) in (..Liste..) repeat (..Anweisungen...);
```

Deren Bedeutung ist offensichtlich und wird hier nicht formalisiert.

- Der Datentyp `processid` bezeichnet Prozesse; für diese sind Operationen "blockiere" und "deblokiere" definiert, sowie die Funktion "ownpid", die die eigene Prozeßnummer liefert.

Anmerkungen zum nachfolgenden Programm:

Die Warteliste wird hier nur als Menge ohne weitere Struktur behandelt; Tatsächlich ist sie partiell (im Fall $n=1$ sogar total) geordnet. Die Implementierung der Listenoperationen könnte hiervon Gebrauch machen und damit einige der Vergleiche $s.e \geq y$ einsparen.

Im Fall der totalen Ordnung ergibt sich das fast von selbst, aber auch im allgemeinen Fall könnte eine Ersparnis z.B. durch verzeigerte Listen erreicht werden; doch das wird hier nicht weiter untersucht.

Der Aufruf der Prozedur `reduce` muß nicht bei jeder Aktion ANFO/FREI erfolgen, sondern kann von der Überschreitung von Grenzwerten abhängig gemacht werden. Daß diese Reduzierung zulässig ist und das Verhalten des Mechanismus nicht ändert, sieht man leicht ein:

Die Zustände vor und nach einer Reduzierung unterscheiden sich nur durch eine additive Konstante; gegen solche Addition (auf beiden Seiten !) sind aber die Fortsetzungsbedingungen

$$s.e \leq y$$

invariant.

```
module ANFOFREIGENERAL;
```

```
type nonnegative is integer range 0..maxint;  
type nnvector is array (1..n) of nonnegative;
```

```
(*----- private Daten des Moduls -----*)
```

```
s : record e,v : nnvector end record; (* Zähler *)
```

```
li: list of record yy: nnvector, pn: processid end record;
```

```
(* Die Liste li enthält alle Informationen über wartende Prozesse, *)  
(* nämlich Wartebedingung yy und Prozeßnummer pn. *)
```

```

(*----- Initialisierung -----*)
s.e := (Anfangswert); s.v := (0,...,0); li := emptylist;

(*----- lokale Prozedur -----*)
(* zur Überlauf-Vermeidung *)
procedure reduce;
  d : nnvector;
  begin d(*) := min(s.e(*), s.v(*));
    (* alle Variable werden um den Vektor d vermindert *)
    s.e := s.e - d;
    s.v := s.v - d;

    (* auch die Warteliste muß um d reduziert werden *)
    for (yy, pn) in li
      repeat
        remove(li,(yy,pn));
        yy := yy - d;
        insert(li,(yy,pn));
      end repeat;
    end reduce;

(*----- Anforderung -----*)
procedure ANFO(k: nnvector);
  y : nnvector; (* lokale Größe *)
  begin
    reduce;
    s.v := s.v + k;
    y(*) := s.v(*) * sgn(k(*));

    if not (s.e ≥ y)
      then
        insert(li,(y,ownpid));
        blockiere (ownpid);
      endif;
    end ANFO;

(*----- Freigabe -----*)
procedure FREI(k: nnvector);
  begin
    reduce;
    s.e := s.e + k;

    (* welche Anforderungen können deblockiert werden ? *)
    for (yy, pn) in li
      repeat
        if s.e ≥ yy
          then
            remove (li, (yy,pn));
            deblockiere (pn);
          endif;
        end repeat;
    end FREI;

end module;

```

5. V-Systeme

Nachdem wir den neuen, schedule-bezogenen Simulationsbegriff auf F-Systeme angewandt haben, durch die er ursprünglich motiviert wurde, wollen wir in diesem Abschnitt untersuchen, wieweit er Aussagen über V-Systeme erlaubt. Insbesondere ist es von Interesse, die bisherige Theorie, d.h. den auf Lipton zurückgehenden Simulationsbegriff aus der Sicht von Schedules zu beschreiben.

Es sei daran erinnert, daß wir nicht eine starre Definition von "Simulation" gewählt haben, sondern durch Wahl verschiedener Schedule-Klassen eine Vielzahl von Varianten ermöglichen.

Es geht also in diesem Abschnitt um die Frage, ob durch geeignete Wahl der Scheduleklasse unser Simulationsbegriff, eingeschränkt auf den Fall der V-Systeme, mit dem Lipton'schen äquivalent wird.

Allerdings ist dabei einschränkend zu bemerken:

V-Systeme sind nicht exakt das gleiche wie die von Lipton betrachteten Prozeßsysteme. Der Unterschied besteht in der expliziten Betrachtung des Aufrufzeitpunkts (f-call) einer Aktion. Zwischen Lipton-Prozeßsystemen und V-Systemen besteht folgende enge Beziehung:

Aus einem gegebenen Lipton-System entsteht ein V-System durch Hinzufügen der Aktion f-call für jede Aktion f.

Aus einem gegebenen V-System entsteht ein Lipton-System durch Weglassen aller call-Aktionen.

In Abschnitt 3.6. wurde die Eigenschaft "explizit-blockierend" definiert, nämlich daß ein nicht (explizit) blockierter Prozeß immer eine Aktion ausführen kann. Genau das gilt für Lipton-Systeme nicht, weil dort die Blockaden "implizit" sich aus den Vorbedingungen ergeben. Gewissermaßen ist das V-System der "explizit-blockierende Abschluß" eines gegebenen Lipton-Systems.

Vor der Betrachtung des Lipton'schen Begriffs müssen wir unsere eigene Definition aus 3.4. noch etwas modifizieren.

5.1. Neudefinition der Simulation

Der Simulationsbegriff aus Abschnitt 3.4. berücksichtigt die Rolle der call-Aktionen in V-Systemen nicht ausreichend: Diese Aktionen sind ja eine Hilfskonstruktion und haben keine Wirkung auf den seriellen Zustand des Prozeßsystems.

Dabei sind mit "serieller Zustand" alle eigentlichen Variablen gemeint, einschließlich der Semaphore; der parallele Zustand wäre dann diejenige Information, die zur Verwaltung des Synchronisationsverhaltens benötigt wird, nämlich die Prozeßzustände NB und die Vorbedingungen als Prädikate $Pr_1, \dots, Pr_n$.

Beim Simulationsbegriff geht diese Zweiteilung des Zustands in der Weise ein, daß die Forderungen

$$r(\alpha) = \beta$$

sich auf den seriellen,
die Relationen

$$sf(\alpha) \text{ wk } sf(\beta) \quad \text{bzw.} \quad sf(\beta) \text{ wk } sf(\alpha)$$

auf den parallelen Zustand beziehen.

Nun sollte man aber zwei Folgen wie

f1-exec f2-call f3-exec f2-exec

und

f2-call f1-exec f3-exec f2-exec

als gleich ansehen, wenn nur das nach außen beobachtete Verhalten, die seriellen Zustände, und nicht das Blockadeverhalten zur Diskussion stehen.

Das führt aber dazu, daß die Realisierungsfunktion die call-Aktionen ignorieren muß; es ist auch wenig sinnvoll,

$$r(f1\text{-call}) = f2\text{-exec}$$

oder ähnliches zu definieren.

So kommt man zur folgenden Definition, die für F-Systeme mit der Definition aus 3.4. übereinstimmt.

Definition:

Sei $P=(A,D,w)$ ein Prozeßsystem.

Die Abbildung

$$\text{netto} \in \text{Abb}(\text{Star}(A), \text{Star}(A))$$

ist wie folgt definiert:

$$\text{netto}(f\text{-call}) := \text{lambda},$$

$$\text{netto}(f\text{-exec}) := f\text{-exec};$$

Anschaulich ist diese Abbildung nichts als das Weglassen der call-Aktionen, insbesondere ist für ein F-System

$$\text{netto} = \text{Identität}.$$

Die Wertemenge $\text{netto}(\text{Star}(A))$ ist selbst, wie $\text{Star}(A)$, eine Halbgruppe bezüglich der Aneinanderreihung.

Nun soll sich die Realisierungsfunktion im Fall von V-Systemen nicht mehr auf sämtliche, sondern nur auf die "echten", d.h. netto-Aktionen beziehen:

Definition:

Seien $P=(A_1,D_1,w_1)$, $Q=(A_2,D_2,w_2)$ Prozeßsysteme mit n Prozessen.

Eine Abbildung

$$r \in \text{Abb}(\text{netto}(\text{Star}(A_2)), \text{netto}(\text{Star}(A_1)))$$

heißt Realisierung von P durch Q , falls r ein Halbgruppen-Homomorphismus und strukturerhaltend ist.

Daraus ergibt sich die nächste

Definition:

Sei $K=(M,w_k)$ eine Scheduleklasse und r eine Realisierung von P durch Q .

Q simuliert P bezüglich der Klasse K durch r , wenn gilt

(1) r ist vollständig bezüglich K , d.h.

für jedes $\beta \in \text{act}(P)$ mit $\text{sf}(\beta) \in M$

gibt es ein $\alpha \in \text{act}(Q)$,

so daß $r(\text{netto}(\alpha)) = \text{netto}(\beta)$

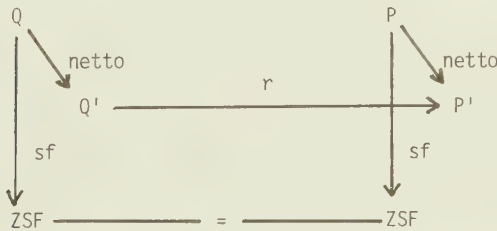
und $\text{sf}(\alpha) \text{ wk } \text{sf}(\beta)$.

- (2) r ist sicher bezüglich K , d.h.
 für jedes $\alpha \in \text{act}(Q)$ mit $\text{sf}(\alpha) \in M$
 gibt es ein $\beta \in \text{act}(Q)$,
 so daß $r(\text{netto}(\alpha)) = \text{netto}(\beta)$
 und $\text{sf}(\beta) \text{ wk } \text{sf}(\alpha)$.

An der Forderung bezüglich der Schaltfolgen hat sich nichts geändert, die Einführung der Funktion netto trägt der besonderen Rolle der call-Aktionen Rechnung.

Die Definitionen dieses Abschnitts gelten nicht nur für den Vergleich von V-Systemen untereinander, sondern auch für den Vergleich von V- mit F-Systemen in Kapitel 6.

Folgende Abbildung veranschaulicht die Definitions- und Wertemengen der vorkommenden Funktionen.



Dabei bedeuten P' und Q' die Prozeßsysteme, die durch Weglassen der call-Aktionen entstehen, und ZSF die Menge der zulässigen Schaltfolgen.

5.2. Blockadefreie Simulation

Um Lipton's Konzept der blockadefreien Simulation mit unseren Begriffen zu erfassen, definieren wir eine entsprechende Scheduleklasse.

Definition:

Die Scheduleklasse $BF_n = (M, wk)$ ist definiert durch

$$M = ZSF_n,$$

$$\alpha \quad wk \quad \beta \iff NB(\alpha) \subset NB(\beta),$$

wobei wie in 3.6. $NB(\alpha)$ die Menge der nach der Schaltfolge α nicht blockierten Prozesse bedeutet.

Die Relation wk bezieht sich also nur auf das letzte Folgenglied, zwei Schaltfolgen gelten als äquivalent, falls am Ende die gleichen Prozesse blockiert sind; der vorherige Verlauf spielt dann keine Rolle.

Die Menge M der zu betrachtenden Schedules ist hier - wie schon in anderen Beispielen - die größtmögliche, nämlich die Menge ZSF_n aller zulässigen Schaltfolgen von n Prozessen.

Unmittelbar aus dieser Definition folgt die in 3.6. als "reichhaltig" bezeichnete Eigenschaft, so daß für Simulation bezüglich der Klasse BF_n das Fundamentaltheorem gelten muß. Zugleich ist wk die schwächste Relation, für die die Klasse (ZSF_n, wk) reichhaltig ist.

Die etwas strengere Voraussetzung für unser Invarianztheorem gilt für BF_n nicht.

Die Simulation bezüglich dieser Klasse stimmt nun mit der blockadefreien Simulation überein:

BF-Theorem:

Seien P' und Q' zwei Prozeßsysteme im Sinne von Lipton (LIPT74), P und Q die zugeordneten V-Systeme, wie anfangs von Kapitel 5. beschrieben.

Sei die Anzahl der Prozesse in beiden Systemen gleich n .

Sei r eine Realisierung von P durch Q .

Dann gilt:

(A) Q' simuliert P' durch r vollständig, sicher und blockadefrei

genau dann wenn

(B) Q simuliert P durch r bezüglich der Klasse Bfn vollständig und sicher.

Dabei ist zu beachten, daß die Realisierung r die gleiche ist, da sie im Fall der V-Systeme sich nur auf die exec-Aktionen erstreckt; diese sind aber dieselben wie im Lipton'schen System.

Zur konzisen Formulierung des Beweises verabreden wir noch zwei Bezeichnungen:

Ein Prozeß P_i heißt nach einem Ablauf α explizit blockiert, wenn er blockiert ist im Sinn von unserer Definition des Prozeßsystems, wenn also gilt $i \notin NB(\alpha)$.

Ein Prozeß $P'(i)$ heißt implizit blockiert, wenn er im Sinne Lipton's blockiert ist, wenn also gilt: für keine Aktion f ist

$$bz(f) = i \quad \text{und}$$

$$\alpha \ f \in act(P').$$

Dann gilt folgendes einfache

Lemma:

Ist $P(i)$ nach einem Ablauf α explizit blockiert, dann ist $P'(i)$ nach dem Ablauf $netto(\alpha)$ implizit blockiert.

Die Umkehrung gilt nicht, aber

ist $P'(i)$ nach einem Ablauf $netto(\alpha)$ implizit blockiert, dann ist entweder

$P(i)$ nach dem Ablauf α explizit blockiert

oder

der Ablauf $\alpha \ f\text{-call}$ ist aktiv in P für eine Aktion f
mit $r(f) = i$.

Beweis: klar aus den Definitionen.

Beweis des Theorems:

Wegen der Voraussetzung gleicher Prozeß-Anzahlen in P und Q ist Lipton's Forderung der Prozeßtreue gleichwertig mit unserer Forderung nach Strukturhaltung. Nun ist zu zeigen:

(A) $\Rightarrow$ (B):

Angenommen Q' simuliere P' wie vorausgesetzt.

a) r ist vollständig bezüglich BF:

Sei $\alpha \in \text{act}(P)$ ein aktiver Ablauf in P. Zu zeigen ist: es gibt ein beta aktiv in Q, so daß

(1) $r(\text{netto}(\beta)) = \text{netto}(\alpha)$ und

(2) $\text{sf}(\beta) \text{ wk } \text{sf}(\alpha)$.

Der Ablauf alpha selbst enthält möglicherweise call-Aktionen, für die Gleichung (1) können diese aber außer acht gelassen werden.

Konstruiere also beta wie folgt. Der Ablauf $\text{netto}(\alpha)$ ist aktiv in P'. Wegen der Vollständigkeit der Simulation von P' durch Q' existiert zu dem Ablauf $\text{netto}(\alpha)$ ein aktiver Ablauf beta1 in Q', mit

$$r(\beta_1) = \text{netto}(\alpha).$$

Sei beta1 ein maximaler solcher Ablauf, d.h. nicht durch unbeobachtbare Aktionen fortsetzbar.

beta1 enthält keine call-Aktionen, ist aber auch in Q aktiv, da nur Aktionen mit wahrer Vorbedingung vorkommen. beta1 enthält aber auch keinerlei (explizite) Blockierungen; um die Ungleichung (2) zu erreichen, wird daher beta1 mit call-Aktionen fortgesetzt, soweit nötig:

Sei $i \in \{1, \dots, n\} - \text{NB}(\alpha)$, d.h. Prozeß P_i ist nach alpha blockiert; dann muß beta so konstruiert werden, daß auch Q_i nach beta blockiert ist. Setze daher

$$\beta = \beta_1 \text{ f(i1)-call f(i2)-call } \dots \text{ f(ii)-call,}$$

wobei

$$\{i_1, i_2, \dots, i_i\} = \{1, \dots, n\} - \text{NB}(\alpha)$$

die Menge der nach alpha blockierten Prozesse, und $f(i)$ die im Prozeß Q_i nach dem Ablauf beta1 anstehende Aktion ist.

Es ist noch zu zeigen, daß β wirklich aktiv ist, d.h. daß im Prozeß Q_i nach dem Ablauf βa_1 die anstehende Aktion blockiert ist.

Wäre dies nicht der Fall, dann wäre (etwa für $i=i_1$)

$\beta a_1 \quad f(i_1)\text{-exec}$

aktiv; wegen der Maximalität von βa_1 müßte dann $f(i_1)$ eine beobachtbare Aktion sein, also

$r(\beta a_1 \quad f(i_1)) = \text{netto}(\alpha) \quad a$, mit einer Aktion a in P' .

Diese Aktion muß wegen der Strukturerhaltung zum Prozeß $P'(i_1)$ gehören. Dies widerspricht aber der Annahme

$i_1 \notin \text{NB}(\alpha)$;

d.h. der Prozeß $P(i_1)$ ist nach α explizit blockiert.

Nach dem Ablauf β sind aber zumindest die Prozesse

$Q(i_1), Q(i_2), \dots, Q(i_i)$

blockiert; somit ist gezeigt

$\text{NB}(\beta) \subseteq \text{NB}(\alpha)$,

und daher nach Definition der Klasse BF:

$\text{sf}(\beta) \text{ wk } \text{sf}(\alpha)$.

Auch die Beziehung (1) folgt aus der Konstruktion.

b) r ist sicher bezüglich der Klasse BF:

Sei β ein aktiver Ablauf in Q . Es ist ein aktiver Ablauf α in P zu konstruieren, so daß gilt:

(3) $r(\text{netto}(\beta)) = \text{netto}(\alpha)$, und

(4) $\text{sf}(\alpha) \text{ wk } \text{sf}(\beta)$.

Sei nun $\alpha a_1 := r(\text{netto}(\beta))$; dies ist wegen der Sicherheit der Simulation von P' durch Q' ein aktiver Ablauf in P' .

Wie oben enthält dieser aber keine Blockaden, insbesondere gilt i.a. nicht $\text{sf}(\alpha a_1) \text{ wk } \text{sf}(\beta)$.

Wir setzen wie oben αa_1 fort durch call-Aktionen:

$\alpha \quad := \quad \alpha a_1 \quad g(i_1)\text{-call } g(i_2)\text{-call } \dots \quad g(i_i)\text{-call},$

wobei $g(i)$ die im Prozeß P_i nach dem Ablauf α anstehende Aktion ist, und $i_1, \dots, i_i$ all die Prozesse durchläuft, die nach dem Ablauf αa_1 keine exec-Aktion ausführen können. Dann ist α jedenfalls aktiv in P . Auch die Gleichung (3) gilt selbstverständlich, weil

$\text{netto}(\alpha) = \text{netto}(\alpha a_1) = \alpha a_1 = r(\text{netto}(\beta))$.

Es bleibt also (4) zu zeigen:

D.h. nach Definition von wk ist zu zeigen: $NB(\alpha) \subseteq NB(\beta)$.

Dies zeigen wir indirekt: sei $i \in \{1, \dots, n\}$ und

$$i \notin NB(\beta).$$

Letzteres heißt: Prozeß Q_i ist nach β (explizit) blockiert. Aus der Blockadefreiheit (und dem Lemma) folgt dann, daß das Bild dieses Prozesses in P' , der Prozeß $P'(i)$, ebenfalls zumindest implizit blockiert ist nach dem Ablauf α_1 . Die Konstruktion von α garantiert aber gerade, daß jeder implizit blockierte Prozeß auch explizit blockiert ist. Somit ist

$$i \notin NB(\alpha).$$

Damit ist die Richtung " $(A) \Rightarrow (B)$ " gezeigt.

$(B) \Rightarrow (A)$:

Angenommen Q simuliere P wie vorausgesetzt. Zu zeigen ist: Q' simuliert P' sicher, vollständig und blockadefrei.

a) r ist vollständig als Realisierung von P' durch Q' :

Dies folgt direkt aus der Vollständigkeit der Realisierung von P durch Q ; sei nämlich β ein aktiver Ablauf in P' , dann ist β auch aktiv in P , und es gibt einen Ablauf α_1 in Q mit

$$r(\text{netto}(\alpha_1)) = \text{netto}(\beta) = \beta.$$

Setze dann $\alpha := \text{netto}(\alpha_1)$; dies ist ein aktiver Ablauf in Q' .

b) r ist sicher als Realisierung von P' durch Q' :

Dies folgt ebenfalls direkt aus der Sicherheit der Realisierung von P durch Q ; sei nämlich α ein aktiver Ablauf in Q' , dann ist α auch aktiv in Q , und es gibt einen Ablauf β_1 in P mit

$$r(\text{netto}(\alpha)) = \text{netto}(\beta_1).$$

Setze dann $\beta := \text{netto}(\beta_1)$; dies ist ein aktiver Ablauf in P' .

c) r ist blockadefrei:

Sei α ein aktiver Ablauf in Q' , und sei der Prozeß $Q'(i)$ nach α implizit blockiert; dann ist zu zeigen: der Prozeß $P'(i)$ ist nach dem

Ablauf $r(\alpha)$ implizit blockiert.

Sei nun $f(i)$ die im Prozeß $Q'(i)$ nach dem Ablauf α anstehende Aktion. Implizite Blockade bedeutet dann: der Ablauf

$\alpha_1 := \alpha \ f(i)\text{-call}$

ist aktiv in Q , und

(5) $i \notin NB(\alpha_1)$.

Aus der Sicherheit der Simulation von P durch Q bezüglich der Klasse BF_n folgt dann:

es gibt einen aktiven Ablauf β_1 in P , so daß

(6) $r(\text{netto}(\alpha_1)) = \text{netto}(\beta_1)$ und

(7) $sf(\beta_1) \text{ wk } sf(\alpha_1)$.

Letzteres heißt $NB(\beta_1) \subseteq NB(\alpha_1)$, also wegen (5):

$i \notin NB(\beta_1)$.

Der Prozeß $P(i)$ ist also nach dem Ablauf β_1 explizit blockiert; dann ist aber der Prozeß $P'(i)$ nach dem Ablauf

$\text{netto}(\beta_1)$

implizit blockiert, und andererseits gilt

$\text{netto}(\beta_1) = r(\text{netto}(\alpha_1)) = r(\alpha)$.

Also ist $P'(i)$ nach dem Ablauf $r(\alpha)$ implizit blockiert.

Q.E.D.

Anmerkungen:

1. Ein Vergleich der beiden Simulationsbegriffe ergibt folgende Entsprechungen:

Lipton:		Klasse BF :
Vollständigkeit	=	Vollständigkeit
Sicherheit		
+	=	Sicherheit.
Blockadefreiheit		

2. Die Voraussetzung $M=ZSF_n$ (aus der Definition $BF_n=(M,w_k)$) geht wesentlich in den Beweis ein: sie bedeutet ja, daß alle Schaltfolgen zulässig sind, also Prozeßwechsel an jeder Stelle erlaubt ist.

Nur dadurch ist es möglich, daß die konstruierten Abläufe alle aktiv sind; insbesondere muß im V-System angenommen werden, daß auch Abläufe der Form $netto(\alpha)$, also solche ohne call-Aktionen aktiv sind. Das bedeutet aber, daß der Scheduler jeweils einem Prozeß den Prozessor entzieht, bevor dieser sich explizit blockieren kann.

Dies soll an einem Beispiel verdeutlicht werden:

```
semaphore s1,s2 init (0);
process P1:  V(s1); P(s2);
process P2:  P(s1); V(s2);
```

Dieses Prozeßsystem erlaubt folgende Abläufe:

(P1:V(s1)-exec) (P1:P(s2)-call) (P2:P(s1)-exec) ...

aber auch

(P1:V(s1)-exec) (P2:P(s1)-exec) ...

Im zweiten Fall ist der Prozeß P1 nicht explizit (wohl aber implizit !) blockiert, weil ihm der Prozessor entzogen wird, bevor er die Aktion P(s2) aufruft.

6. Vergleich von F- und V-Systemen

Die in dieser Arbeit - wie in anderen Arbeiten über Prozeßkoordinierung - definierten Prozeßsysteme sind Abstraktionen von realen Systemen paralleler Programme. Sie dienen als Grundlage für verifizierbare Aussagen über das Verhalten von solchen Systemen, sowie zur Spezifikation von deren Eigenschaften. Da nun zweierlei Ansätze zur Verfügung stehen, stellt sich die Frage, ob beide gleich gut für diese Aufgaben geeignet sind, genauer mehrere Fragen wie:

lassen sich mit den Ansätzen "F-System" und "V-System" die gleichen Verhaltensweisen beschreiben ?

Kann ein abstrakt als V-System gegebenes Problem ebenso durch ein F-System beschrieben (= gelöst) werden ? Und umgekehrt ?

Versuche, eine Methode zur Spezifikation und Implementierung gerade von parallelen Programmen auf einem exakten formalen Modell aufzubauen, gibt es bereits einige (vgl. Abschnitt 1.2). Angesichts der Komplexität paralleler Abläufe erscheint dieser Weg nicht nur vielversprechend, sondern eigentlich zwingend notwendig.

Nun bauen aber die bekannten Modelle zumeist auf dem Grundgedanken des V-Systems auf, was die Spezifikationen betrifft: es werden Vorbedingungen für die einzelnen Aktionen angegeben.

Implementierungen dagegen funktionieren, wie ich schon in Abschnitt 2. zu zeigen versucht habe, im Grunde eher wie F-Systeme.

Aus diesen Gründen ist die Untersuchung der obigen Fragen nicht nur für sich selbst von Interesse, sondern Voraussetzung für formale Spezifikationsmethoden, und damit für die Entwicklung zuverlässiger paralleler Programme überhaupt.

Untersucht man Fragen wie die oben formulierten mittels unserer formalen Begriffe, dann kann man zunächst zwischen unterschiedlichen Scheduleklassen, d.h. zwischen unterschiedlich strengen Simulations- (d.h. hier Lösungs-) begriffen wählen. Mit einem strengeren Simulationsbegriff, d.h. mit einer Scheduleklasse, die vielerlei Schedules umfaßt, wird man feinere Unterschiede feststellen als mit einer kleinen Scheduleklasse.

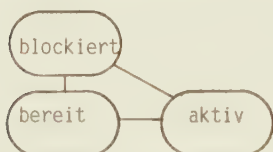
Diese Wahl der Scheduleklasse ist keine theoretische Spielerei, sondern steht in direktem Bezug zu den Überlegungen eines Programmierers: es geht einfach um die Frage bei der Konzipierung und Implementierung eines portablen Programms, welche Annahmen über das Betriebssystem gemacht werden. Je schwächer diese Annahmen (in diesem Fall über die Scheduling-Strategie), desto größer ist die Scheduleklasse und der Geltungsbereich der Implementierung.

6.1. Zustandsübergänge

Bevor wir F- und V-Systeme auf formaler Basis vergleichen, untersuchen wir die Übergänge zwischen den verschiedenen Prozeßzuständen für die beiden Systemtypen.

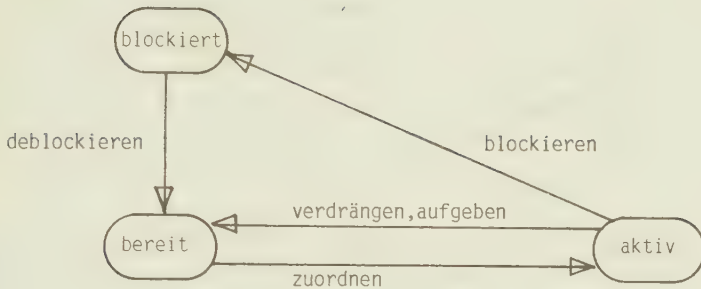
Betriebssysteme kennen unterschiedliche Einteilungen der Prozesse in Zustände wie laufend (aktiv, rechnend), wartend (blockiert), bereit, wartend auf Transferende, wartend auf Hauptspeicherkachel, wartend auf Auftrag, suspendiert, markiert, dringend, etc.

Für theoretische Untersuchungen beschränken wir uns hier auf das "klassische" Zustandsdreieck:



In unserem formalen Modell kommt zwar nur "blockiert" und "nicht blockiert" vor, aber die Einteilung der nicht blockierten Prozesse in die Zustände "bereit" und "aktiv" ergibt sich in natürlicher Weise: als aktiv wird immer derjenige angesehen, der gerade eine Aktion ausführt; dabei bringt es die Sichtweise der Abläufe als Folgen mit sich, daß zu jedem Zeitpunkt höchstens ein Prozeß aktiv ist, was dem Ein-Prozessor-Fall entspricht.

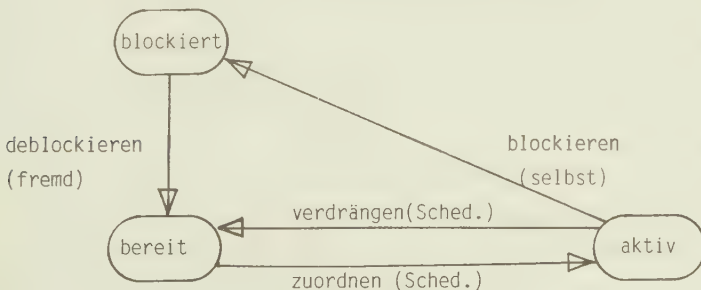
Welche Übergänge zwischen diesen drei Zuständen kommen nun vor? Jede Vorlesung und jedes Lehrbuch über Betriebssysteme gibt darauf etwa die folgende Antwort (nach (WETT78)):



Dies trifft nicht nur auf reale Betriebssysteme, sondern auch auf F-Systeme recht gut zu:

Insbesondere kann in einem F-System der aktive Prozeß sich selbst blockieren, oder (vom Scheduler) verdrängt werden; Prozessoraufgabe ist nicht vorgesehen. Ein bereiter Prozeß kann (vom Scheduler) zugeordnet, und damit im nächsten Schritt aktiv werden; er kann jedoch nicht blockiert werden, ohne selbst zuvor aktiv zu sein. Ein blockierter Prozeß kann (durch Erfüllung seiner Fortsetzungsbedingung) deblockiert werden.

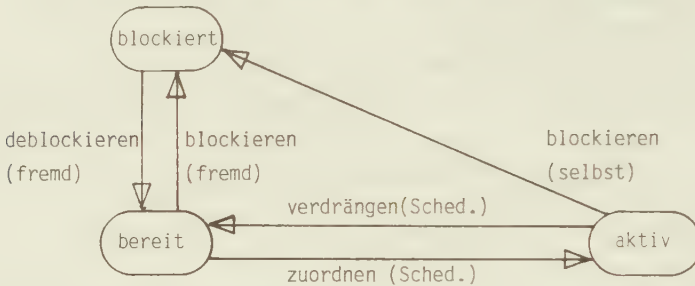
Für F-Systeme läßt sich also das Zustandsdiagramm so zeichnen: (In Klammern ist der Urheber des betreffenden Übergangs genannt, nämlich ein fremder Prozeß, der eigene Prozeß oder der Scheduler.)



Anders ist die Situation für V-Systeme:

Dort ist es zusätzlich zu den obigen Übergängen auch möglich, daß ein bereiter Prozeß durch einen fremden blockiert wird; wenn nämlich eine wahre Vorbedingung falsch wird, wird der betreffende Prozeß blockiert.

Das Zustandsdiagramm für V-Systeme sieht demnach so aus:



Dabei braucht man nicht zwischen impliziter und expliziter Blockierung zu unterscheiden.

Die Beobachtung, daß das Zustandsdiagramm für V-Systeme sich von demjenigen für reale Betriebssysteme nicht unwesentlich unterscheidet, bedeutet nicht, daß man V-Systeme nicht implementieren könnte; es legt aber nahe, daß dies schwieriger sein wird als die Implementierung von F-Systemen. Dies stimmt mit den Resultaten von Abschnitt 2. überein.

An der Blockierung bereiter Prozesse zeigt sich auch ein wichtiger Unterschied in der Auffassung der Wartebedingung:

- Vorbedingungen müssen in dem Moment gelten, wenn die Aktion ausgeführt wird; wenn also eine solche Bedingung eintritt, der betreffende Prozeß aber nicht zugeordnet wird, muß die Bedingung weiter überwacht werden.
- Fortsetzungsbedingungen müssen dagegen nur "irgendwann einmal" gelten, von da an ist die betreffende Aktion als erledigt zu betrachten, so daß die Bedingung nicht mehr ausgewertet zu werden braucht.

6.2. Zeitfreie Simulation

In diesem Abschnitt vergleichen wir die beiden Konzepte "F- und V-System" bezüglich der Scheduleklasse ZF der "zeitfreien" Schedules, genauer der Schedules ohne Zeitscheibenunterbrechung. Es wird also vorausgesetzt, Prozeßwechsel fänden nur statt, wenn mindestens ein Prozeß blockiert oder deblockiert wird.

Wir beweisen zwei Sätze über die Simulation eines gegebenen F-Systems durch ein auf einfache Weise konstruiertes V-System und umgekehrt. Dabei müssen wir außer der Beschränkung auf zeitfreie Schedules noch weitere Eigenschaften des gegebenen Prozeßsystems voraussetzen.

6.2.1. Voraussetzungen

Dieser Abschnitt soll die Voraussetzungen schaffen für die angekündigten Simulationssätze.

Zunächst definieren wir die Scheduleklasse ZFn der zeitfreien Schedules. Gegenüber anderen bisher betrachteten Scheduleklassen ist sie vor allem dadurch gekennzeichnet, daß die Menge M, der größte Schedule der Klasse, hier eingeschränkt ist; darin drückt sich aus, daß Prozeßwechsel nicht an jeder Stelle möglich sind. Die zugehörige Menge M wurde schon in Abschnitt 3.3. beispielhaft vorgestellt; die Relation wk wird hier den Erfordernissen des Beweises entsprechend neu definiert.

Definition:

Die Relation wk1 auf Schaltfolgen von n Prozessen ist der transitive Abschluß folgender Axiome:

(1) $\alpha \text{ wk } \alpha$

(2) $NB(\alpha) = NB1 \Rightarrow$

$(\alpha(i, NB1)(i, NB2)) \text{ wk } (\alpha(i, NB2))$

(streng nicht-zählend)

(3) $(i, NB) \text{ wk } (i, NB)(j, NB)$

(Adsorption)

(4) $(\beta_1 \text{ wk } \beta_2) \Rightarrow ((\alpha \beta_1 \gamma) \text{ wk } (\alpha \beta_2 \gamma))$
 (Verträglichkeit mit der Monoid-Struktur)

(5) $\text{NB}_1 \subseteq \text{NB}_2 \Rightarrow (i, \text{NB}_1) \text{ wk } (i, \text{NB}_2)$ (Monotonie)

Die Axiome (1),(3),(4) sind von der Klasse SNZn bekannt; (2) wurde etwas abgeändert und wird am besten verständlich, wenn man es in der - formal nicht einwandfreien - Form

$(\dots, \text{NB}_1)(i, \text{NB}_1)(i, \dots) \text{ wk } (\dots, \text{NB}_1)(i, \dots)$

schreibt - die anschauliche Bedeutung bleibt: wo ein Prozeß aktiv sein kann, da darf er beliebig viele Aktionen ausführen, solange er keine Blockade ändert.

Axiom (5) besagt: ist eine Schaltfolge α unter einem Schedule möglich, dann gilt das auch für eine Folge α' , die durch Verkleinerung der Nicht-Blockiert-Menge an einer beliebigen Stelle entsteht - natürlich nur sofern α' überhaupt eine zulässige Schaltfolge ist.

Die Relation wk_1 benötigen wir für Abschnitt 6.2.3. (in Verbindung mit beliebigem M), aber auch für die folgende

Definition:

Die Scheduleklasse $\text{ZF}_n = (M, \text{wk})$ ist definiert durch folgende Axiome:

(1) $M =$
 $\{ (p_1, \text{NB}_1)(p_2, \text{NB}_2) \dots (p_k, \text{NB}_k) \in \text{ZSF}_n:$
 für $i=1, 2, \dots, k-2$ gilt:
 $p(i+1) = p(i+2)$ or $\text{NB}_i \neq \text{NB}(i+1)$
 $\};$

(2) $\text{wk} = \text{wk}_1$.

Im allgemeinen kann eine synchronisierende Aktion in einem F-System sowohl andere Prozesse blockieren wie auch gleichzeitig (bzw. anschließend) sich selbst blockieren. In den meisten vorkommenden Mechanismen besteht aber eine Zweiteilung in dem Sinn: gewisse Aktionen führen potentiell zur Blockierung, andere zur Deblockierung von

Prozessen. Diese Eigenschaft formuliert die folgende

Definition:

Ein Prozeßsystem P heißt bipartit, wenn für jede Aktion a gilt:
Es gibt nicht zwei Abläufe alpha, beta,
so daß

$$\alpha a, \beta a \in \text{act}(P)$$

und

$$NB(\alpha a) = NB(\alpha) - \{bz(a)\}$$

und

$$\text{für ein } j \neq bz(a) \text{ ist: } j \in NB(\beta a) - NB(\beta) .$$

Verboden ist also, daß die Aktion a einmal sich selbst blockiert, ein andermal einen Prozeß Pj deblockiert. Insbesondere ($\alpha=\beta$) darf a nicht gleichzeitig diese beiden Wirkungen haben.

Diese Bedingung ist etwas strenger als die wirklich benötigte: es würde genügen, gleichzeitiges Blockieren und Deblockieren zu verbieten; jedoch erscheint mir der strengere Begriff hier klarer.

Beispiele:

Alle in Abschnitt 4. diskutierten Mechanismen, außer der Lösung des 2-Erzeuger-2-Verbraucher-Problems (Abschnitt 4.1., Operationen PROD1, CONS1, PROD2, CONS2) sind bipartit; schon an den Definitionen sieht man dort die Zweiteilung der synchronisierenden Operationen in blockierende und deblockierende.

Für eine Operation mit der Fortsetzungsbedingung true ist dabei nichts zu zeigen.

Als Beispiel zeigen wir, daß eine ANFO-Operation nie einen Prozeß deblockieren kann:

Fortsetzungsbedingungen sind im ANFO/FREI-System immer von der Form

$$s.e \geq y.$$

Die ANFO-Operation erhöht aber nicht die Größe s.e, und y ist ja eine lokale Variable.

Dagegen widerspricht die Operation

$\text{CONS1} = \text{do } (y := s1.v := s1.v+1) \text{ wait } (s1.e \geq y)$

aus Abschnitt 4.1. der Definition von bipartit, denn in dem zugehörigen Prozeßsystem kommt eine Fortsetzungsbedingung der Form

$(s2.e, s1.v) \leq (y2, y1)$

vor, und CONS1 kann sowohl sich blockieren wie andere (hier die Operation CONS2) deblockieren.

V-Systeme sind dagegen immer bipartit, weil die call-Aktion keine andere Wirkung hat als sich selbst zu blockieren, während die exec-Aktion niemals eine explizite Blockade des eigenen Prozesses herbeiführen kann.

Aus diesen Beispielen wird schon ersichtlich, daß "bipartit" für die Simulation von F-Systemen durch V-Systeme eine notwendige Voraussetzung sein wird; dies wird im nächsten Abschnitt weiter erhärtet.

Die zweite Voraussetzung hängt mit den im letzten Abschnitt diskutierten Zustandsübergängen zusammen. Wie dargelegt, verhalten sich F- und V-Systeme grundverschieden dann, wenn eine zunächst wahre Wartebedingung wieder falsch wird.

Dies auszuschließen, ist der Sinn der folgenden

Definition:

Ein F- oder V-Prozeßsystem P heißt remanent, wenn für jedes Prädikat B, das als Fortsetzungs- oder Vorbedingung einer Aktion

$a = (\text{do } A \text{ wait } B)$ bzw.

$a = (\text{when } B \text{ do } A)$

vorkommt, für jeden aktiven Ablauf alpha, und für jede Aktion b mit $\text{bz}(b) \neq \text{bz}(a)$

gilt:

$B(\text{value}(\alpha)) \Rightarrow B(\text{value}(\alpha \text{ b})),$

d.h. wenn B einmal gilt, kann es von einem fremden Prozeß nicht falsch gemacht werden.

Beispiele:

Ein V-System mit P/V-Operationen ist i.a. nicht remanent, weil die Vorbedingung $s > 0$ einer P-Operation im Falle $s=1$ durch Ausführung einer anderen P-Operation falsch wird.

Auch F-Systeme brauchen nicht immer remanent zu sein, jedoch gilt dies für alle F-Systeme vom Zähler-Typ: Da alle Zählvariablen s monoton wachsen, kann eine Bedingung der Form

$$s \geq y$$

nicht mehr falsch werden, nachdem sie einmal galt.

Damit sind die Voraussetzungen über die Scheduleklasse sowie über das zu simulierende System geschaffen.

Die Sätze der nächsten Abschnitte beziehen sich auf die Simulation eines gegebenen F-Systems nicht durch ein beliebiges V-System (und umgekehrt), sondern durch ein methodisch, und zwar auf die einfachste denkbare Weise konstruiertes. Diese Konstruktionen beschreiben die letzten beiden Definitionen:

Definition:

Sei $P=(A,D,w)$ ein beliebiges F-System. Das V-System $TVS(P) := (A',D,w)$ hat die gleichen Daten und den gleichen Anfangszustand wie P . Jede Aktion

$(a: \text{do } X \text{ wait } B);$

in P wird in $TVS(P)$ widergegeben durch die Aktionenfolge

$(a-x: \text{when true do } X); \quad (a-w: \text{when } B \text{ do skip});$

skip steht für die leere Aktion. Es ist offensichtlich, wie sich die Wirkungen der beiden Aktionen auf Befehlszähler und Variable aus den Wirkungen von a ergeben.

Die Zusätze "-x" (execute) und "-w" (wait) zu dem Namen (hier a) der Aktion dienen zur Unterscheidung der beiden Teile.

Um ein V-System anzugeben, genügt es seine "netto"-Aktionen anzugeben; es versteht sich von selbst, daß hier zwischen

a-x-exec, a-w-call und a-w-exec

zu unterscheiden ist - a-x-call kommt nicht vor, wegen der Vorbedingung "true".

Definition:

Sei $P=(A,D,w)$ ein beliebiges V-System. Das F-System $TFS(P) :=(A',D,w)$ hat die gleichen Daten und den gleichen Anfangszustand wie P. Jede Aktion

(a: when B do X);

in P wird in $TFS(P)$ wiedergegeben durch die Aktionenfolge

(a-w: do skip wait B); (a-x: do X);

Die Bezeichnungen TVS und TFS sollen auf ein "in trivialer Weise konstruiertes V-/F-System" hinweisen.

6.2.2. V-System simuliert F-System

Satz:

Sei $ZF_n = (M, wk)$ wie in 6.2.1. definiert.

Sei P ein bipartites und remanentes F-System. Dann simuliert $Q := TVS(P)$ P bezüglich der Klasse ZF_n durch r , wobei r die naheliegende Realisierung

für alle Aktionen a in P :

$r(a-x) = a$; $r(a-w) = \lambda$;

ist.

Bemerkung:

Angesichts der Beispiele in Abschnitt 4. erscheinen die beiden Voraussetzungen "bipartit" und "remanent" nicht zu gravierend, wenn auch keineswegs trivial; dagegen ist die Einschränkung auf zeitfreie Schedules bedeutend - die meisten Betriebssysteme kennen irgendeine Form von Zeitscheibenstrategie !

Auf die Notwendigkeit aller drei Voraussetzungen wird weiter unten eingegangen.

Beweis des Satzes:

Es ist zu zeigen: r ist vollständig und sicher. Da ein V-System beteiligt ist, gilt der Simulationsbegriff aus Abschnitt 5, wo die Funktion netto eingeführt wurde.

a) r ist vollständig bezüglich ZF_n :

Sei β ein aktiver Ablauf in P mit $sf(\beta) \in M$.

Wir müssen einen aktiven Ablauf α in Q finden, so daß

(1) $r(\text{netto}(\alpha)) = \text{netto}(\beta)$ und

(2) $sf(\alpha) \text{ wk } sf(\beta)$.

α besteht dabei aus den Aktionen $a-x\text{-exec}$, $a-w\text{-call}$, $a-w\text{-exec}$ für jede in β vorkommende Aktion a . Die Reihenfolge der beobachtbaren

Aktionen a-x-exec ist durch beta vorgegeben, die anderen müssen an geeigneten Stellen eingefügt werden.

Und zwar muß jeweils a-w-exec irgendwann nach der Aktion a-x-exec vorkommen, jedoch bevor der Prozeß, zu dem a gehört, die nächste Aktion ausführt. Dagegen braucht a-w-call nicht immer vorzukommen. Die Konstruktion wählt für die Aktionen a-w-exec jeweils der spätesten möglichen Zeitpunkt:

Wir konstruieren alpha durch Induktion über die Länge von beta.
Sei also

beta = beta1 (a: do X wait B), mit bz(a) = i;

Induktionshypothese: sei alpha1 schon konstruiert, so daß

r(netto(alpha1)) = netto(beta1) und
sf(alpha1) wk sf(beta1), und darüber hinaus
NB(alpha1) = NB(beta1) .

Letzteres ist mehr als die eigentliche Aussage, wird aber für den Beweis gebraucht.

Setze NB1 := NB(beta1) und NB2 := NB(beta).

Wir definieren alpha in zwei Schritten.

Schritt 1. Einfügen von b-w-exec.

Dieser Schritt liefert als Resultat einen Ablauf alpha2, der sich von alpha1 dadurch unterscheidet, daß - falls nötig - eine Aktion b-w-exec eingefügt wird, deren zugehörige Aktion b schon in beta1 enthalten ist.

Unterscheide folgende Fälle:

Fall 1. in beta1 kommt keine Aktion b vor mit bz(b)=i

D.h. der Prozeß Pi war noch nicht aktiv.

Setze alpha2 := alpha1 .

Fall 2. beta1 = beta11 b beta12 , mit bz(b)=i, und in beta12 kommt keine Aktion b' vor mit bz(b')=i.

Setze dann alpha2 := alpha1 b-w-exec.

alpha2 ist aktiv in Q: Denn in Prozeß Qi steht die Aktion b-w-exec an, und deren Vorbedingung ist erfüllt: letzteres

ist der Fall, weil der Ablauf $\text{beta1 } a$ (mit $\text{bz}(a)=i$) aktiv ist; das bedeutet ja, die Fortsetzungsbedingung von b muß irgendwann gegolten haben; wegen der Eigenschaft "remanent" folgt daraus, daß sie immer noch gilt.

In beiden Fällen ist $\text{NB}(\alpha_2) = \text{NB}(\alpha_1) = \text{NB1}$, da die Aktion $b\text{-w-exec}$ keine Blockierung ändert.

Schritt 2. Repräsentation der Aktion a .

Der Ablauf α_2 wird jetzt fortgesetzt zum Ablauf α , so daß (1) und (2) gelten.

Fall 1. $\text{NB2} = \text{NB1} - \{i\}$

D.h. Prozeß P_i blockiert sich mit der Aktion a . Setze dann $\alpha := \alpha_2 \text{ a-x-exec a-w-call}$.

Dies ist aktiv; denn die Vorbedingung B von $a\text{-w}$ gilt nach dem Ablauf $\alpha_2 \text{ a-x-exec}$

nicht, weil sich der aufrufende Prozeß blockiert.

(1) gilt, weil

$$\begin{aligned} r(\text{netto}(\alpha)) &= r(\text{netto}(\alpha_2)) \text{ r(a-x-exec)} = \\ &= \text{netto}(\text{beta1}) \quad a. \end{aligned}$$

Es bleibt (2) zu zeigen:

Wegen der Voraussetzung "P bipartit" kann die Aktion $a\text{-x-exec}$ keine Blockadeänderung bewirken, und es gilt:

$$\begin{aligned} \text{sf}(\alpha) &= \text{sf}(\alpha_2) (i, \text{NB1}) (i, \text{NB2}) \\ &\quad \left\{ \begin{array}{l} = \text{sf}(\alpha_1) (i, \text{NB1}) (i, \text{NB2}), \\ \qquad \qquad \qquad \text{falls Schritt 1, Fall 1} \end{array} \right. \\ &\quad \text{O D E R} \\ &\quad \left\{ \begin{array}{l} = \text{sf}(\alpha_1) (i, \text{NB1}) (i, \text{NB1}) (i, \text{NB2}), \\ \qquad \qquad \qquad \text{falls Schritt 1, Fall 2.} \end{array} \right. \end{aligned}$$

In beiden Fällen ist aber

$$\begin{aligned} \text{sf}(\alpha) &\text{ wk } \text{sf}(\alpha_1) (i, \text{NB2}) \quad (\text{wegen nicht-zählend,} \\ &\qquad \qquad \qquad \text{beachte } \text{NB}(\alpha_1) = \text{NB1}) \\ &\text{ wk } \text{sf}(\text{beta1}) (i, \text{NB2}) \\ &= \text{sf}(\text{beta}); \end{aligned}$$

Fall 2. $i \in \text{NB}(\beta)$.

D.h. P_i blockiert sich nicht, deblockiert aber möglicherweise andere Prozesse.

Setze dann $\alpha := \alpha_2 \text{ a-x-exec}$.

Dies ist aktiv.

(1) gilt klarerweise.

Die Wirkung der Aktion α wird von a-x-exec vollständig widergegeben, insbesondere ist $\text{NB}(\alpha) = \text{NB}_2$ und daher

$$\text{sf}(\alpha) = \text{sf}(\alpha_2) (i, \text{NB}_2)$$

$$\text{wk sf}(\beta), \quad \text{analog wie oben.}$$

Außer (1) und (2) ergab der Beweis auch

$$\text{NB}(\alpha) = \text{NB}(\beta),$$

so daß die Induktionshypothese im nächsten Schritt noch gilt.

Damit ist die Vollständigkeit gezeigt; die Voraussetzung "zeitfrei" wurde noch nicht verwendet, wohl aber die anderen Voraussetzungen.

Der nicht-triviale Kern des Teils a) läßt sich so zusammenfassen:

Die call-Aktion führt immer dann zur Blockierung eines Prozesses, wenn dessen Fortsetzungsbedingung nicht erfüllt ist; die betreffende Schaltfolge $\text{sf}(\alpha)$ steht in der Relation (2), weil die Teilaktion a-x-exec in diesem Fall keine Blockadeänderung bewirkt (bipartit!). Das "Nachhinken" der Aktion a-w-exec ist möglich wegen der Eigenschaft "remanent".

b) r ist sicher bezüglich ZFn :

Sei α ein aktiver Ablauf in Q mit $\text{sf}(\alpha) \in M$.

(Im Gegensatz zu früheren Beweisen ist letzteres eine echte Einschränkung: für Abläufe, die Prozeßwechsel auch an Stellen ohne Blockadeänderung enthalten, ist hier nichts zu zeigen.)

Dann ist zu zeigen: es gibt einen aktiven Ablauf β in P mit

$$(3) \quad r(\text{netto}(\alpha)) = \text{netto}(\beta) \quad \text{und}$$

$$(4) \quad \text{sf}(\beta) \text{ wk sf}(\alpha).$$

Da P ein F-System, die netto-Funktion also in P die Identität ist, ist

(3) gleichwertig mit:

(3') $r(\text{netto}(\alpha)) = \beta$.

Man erhält also den Ablauf β durch Weglassen aller Aktionen a -w-call und a -w-exec im Ablauf α . Zu zeigen ist: dieses β ist aktiv in P .

Da β in den eigentlichen Aktionen a -x-exec mit α übereinstimmt, ist nur noch zu zeigen, daß in β jeder Prozeß wirklich deblockiert wird, bevor er seine nächste Aktion ausführt. Dies ist der Fall, weil die Aktionen a -w-exec im Ablauf α garantieren, daß die betreffende Fortsetzungsbedingung zumindest zu einem Zeitpunkt gegolten hat; aus der Eigenschaft "remanent" folgt dann, daß sie noch gilt, wenn die nächste Aktion des gleichen Prozesses ausgeführt wird.

Somit ist noch (4) zu zeigen; dies geschieht wieder durch Induktion.

Sei also $\alpha = \alpha_1 f$, $\beta_1 := r(\text{netto}(\alpha_1))$,
 $i := \text{bz}(f)$ und $\text{NB}_1 := \text{NB}(\alpha_1)$.

Induktionshypothese:

es ist $\text{sf}(\beta_1) \text{ wk } \text{sf}(\alpha_1)$

und entweder $\text{NB}(\beta_1) = \text{NB}(\alpha_1)$

oder $\text{NB}(\beta_1) = \text{NB}(\alpha_1) - \{j\}$,

wobei im letzteren Fall α_1 die Form hat

$\alpha_1 = \alpha_1' b\text{-x-exec}$, $\text{bz}(b)=j$.

Unterscheide drei Fälle:

Fall 1. $f = a\text{-x-exec}$

Dann gilt $\beta = \beta_1 a$,

$\text{sf}(\beta) = \text{sf}(\beta_1) (i, \text{NB}_2)$,

$\text{sf}(\alpha) = \text{sf}(\alpha_1) (i, \text{NB}_2')$

Hier gibt es zwei weitere Fälle.

Fall 1.1. $i \in \text{NB}_2$,

d.h. Prozeß P_i blockiert sich nicht. Dann hat die Aktion $a\text{-x-exec}$ die gleiche Wirkung wie a , und es gilt $\text{NB}_2 = \text{NB}_2'$; folglich $\text{sf}(\beta) \text{ wk } \text{sf}(\alpha)$.

Fall 1.2. $i \notin NB2$,

d.h. der Prozeß P_i blockiert sich, und es ist

$$NB2 = NB2' - \{i\},$$

$NB2' = NB1$, da die Aktion $a-x-exec$ sich nicht selbst blockiert, aber auch sonst keine Blockadeänderung bewirkt.

Wegen $NB2 \subset NB2'$ ist dann aber nach den Axiomen für wk auch
 $sf(beta) \quad wk \quad sf(alpha)$.

Fall 2. $f = a-w-call$,

dann gilt $beta = beta1$,

$$sf(alpha) = sf(alpha1) (i, NB2),$$

mit $NB2 = NB1 - \{i\}$. In diesem Fall muß aber a die letzte Aktion von $beta1$ sein; denn a darf keine Prozesse blockieren oder deblockieren, und deshalb kann auf a kein Prozeßwechsel folgen. (zeitfrei !)

Also ist $alpha1 = alpha1' a-x-exec$,

$$sf(alpha) = sf(alpha1') (i, NB1) (i, NB2)$$

$$wk \quad sf(alpha1') (i, NB1) (i, NB1) \quad (NB2 \subset NB1)$$

$$wk \quad sf(alpha1') (i, NB1) \quad (\text{nichtzählend})$$

$$= sf(alpha1)$$

$$wk \quad sf(beta1) = sf(beta)$$

nach Induktionshypothese und den Axiomen für wk .

Außerdem gilt $NB(alpha) = NB(beta)$.

Fall 3. $f = a-w-exec$.

Dann ist ebenfalls $beta = beta1$,

$$sf(beta) = sf(beta1)$$

$$wk \quad sf(alpha1) (i, NB1) \quad (\text{Adsorption, beachte}$$

$$NB(beta1) = NB(alpha1) = NB1)$$

$$= sf(alpha).$$

Auch hier ist $NB(alpha) = NB(beta) = NB1$.

Damit ist in allen Fällen der Induktionsschritt etabliert.

Der Kern von Teil b) liegt in den Fällen 1.2. und 2. - die Voraussetzung "zeitfrei" garantiert, daß im Falle einer nicht erfüllten Fortsetzungsbedingung die zugehörige call-Aktion $a-w-call$ in Q unverzüglich nach der beobachtbaren Aktion $a-x-exec$ aufgerufen wird, und somit das Blockadeverhalten von Q dem von P entspricht.

Q.E.D.

Damit ist der Beweis des Satzes abgeschlossen; im folgenden soll noch die Notwendigkeit der einzelnen Voraussetzungen gezeigt werden.

Dabei betrachten wir jeweils ein F-System P und sein zugeordnetes V-System $TVS(P)$ unter einem bestimmten Schedule. Abschnitt 3.4. lehrte ja, daß die Simulation bezüglich einer Scheduleklasse auch Simulation bezüglich jedes einzelnen Schedules dieser Klasse impliziert. Als Gegenbeispiel für unseren Satz genügt es daher, daß die Behauptung " $TVS(P)$ simuliert P " schon bezüglich eines einzelnen Schedules verletzt ist, wenn man eine der drei Voraussetzungen wegläßt.

1. Notwendigkeit von "remanent"

Das F-System

```
P:  integer x init (0);
    process P1: (a: do skip wait x=1);
    process P2: (b1: x:=1); (b2: x:=0);
```

ist ein einfachstes nicht remanentes Beispiel, da die Aktion $b2$ die Bedingung $x=1$ falsch macht.

Das zugeordnete V-System

```
TVS(P):
    process Q1: (a-x: skip); (a-w: when x=1 do skip);
    process Q2: (b1: x:=1 ); (b2: x:=0);
```

simuliert P nicht einmal bezüglich des einfachen nicht-verdrängenden Schedule (dieser wurde in Abschnitt 2.2 beschrieben; er ist insbesondere zeitfrei); nach dem Ablauf

$a-x \ a-w-call \ b1 \ b2$

ist und bleibt nämlich $Q1$ blockiert, während $P1$ bei dem zugehörigen r-Bild

$a \ b1 \ b2$

deblockiert wird. Daß $Q1$ nach der Aktion $b1$ vorübergehend deblockiert war, kann sich wegen der Annahme "nicht-verdrängend" nicht auswirken.

2. Notwendigkeit von "bipartit"

Das nicht bipartite F-System

```
P:  integer x,y init (0);
    process P1:  (a: do y:=1 wait x=1);
    process P2:  (b1: do skip wait y=1); b2;
    process P3:  c;
```

(hier stehen b2, c für beliebige Aktionen) wird unter einem speziellen zeitfreien Schedule betrachtet:

Bei Änderung von Blockaden teilt der Scheduler nach folgender Regel zu:

- falls P1 blockiert ist, hat P3 Vorrang vor P2,
- sonst hat P2 Vorrang vor P1, und P1 vor P3.
- Es wird jeweils der vorrangige unter den nicht blockierten Prozessen aktiviert.

Dies ist ein deterministischer Schedule. Er erlaubt nur einen einzigen Ablauf:

b1 a c b2.

Das V-System TVS(P) ergibt dagegen unter diesem Schedule den Ablauf

alpha = b1-x b1-w-call a-x b1-w-exec b2 a-w-call c,

mit $r(\alpha) =$

b1 a b2 c.

Die Tatsache, daß mit der Aktion a-x der Prozeß Q2 schon deblockiert, Q1 aber noch nicht blockiert wird, führt hier also zu unterschiedlichem Verhalten gegenüber dem System P.

3. Notwendigkeit von "zeitfrei"

Auch hier genügt wieder die Untersuchung eines sehr einfachen F-Systems unter einem nicht zeitfreien Schedule.

```
P:  integer x ;
    process P1:  (a1: do x:=1 wait x=0); a2
    process P2:  (b1: x:=0 ); b2
```

P ist bipartit und remanent. Als Schedule setzen wir folgende Regel

voraus:

Neuzuteilung ist zu jedem Zeitpunkt möglich (nämlich ausgelöst durch Zeitscheibenende); unter den nicht blockierten Prozessen wird dann, falls vorhanden, einer bevorzugt, der seit seiner letzten Aktion nicht blockiert war. Man kann sich als Hintergrund vorstellen, daß Prozesse, die einmal blockiert waren, vom Hauptspeicher verdrängt werden.

Dann ist im V-System TVS(P) folgender Ablauf aktiv:

alpha = a1-x (*) b1-x b1-w-exec a1-w-exec a2 b2,
wobei an der Stelle (*) ein Zeitscheibenende angenommen wird. An dieser Stelle ist die Aktion a1-x schon ausgeführt, aber Prozeß Q1 noch nicht blockiert. Daher kann er später (a1-w-exec) gleich wieder zugeteilt werden.

Es ist aber $r(\alpha) = a1 \ b \ a2 \ b2$, ein Ablauf, der in P unter dem angenommenen Schedule nicht aktiv ist: weil Prozeß P1 vorher blockiert war, muß nach der Aktion b Prozeß P2 bevorzugt werden.

Zur Illustration des obigen Beweises lohnt es sich, am Beispiel dieses letzten Prozeßsystems P die unter zeitfreiem Schedule möglichen Abläufe zu vergleichen:

Es gibt nur drei Abläufe (die Nichtblockiert-Mengen sind hier ohne Mengenklammern geschrieben).

alpha1	=	a1-x	a1-w-call	b1-x	a1-w-exec	a2	b1-w-exec	b2
sf(alpha1)=		(1,12)	(1,2)	(2,12)	(1,12)	(1,2)	(2,2)	(2,0)
r(alpha1) =		a1		b1		a2		b2
sf(r(alpha1))=		(1,2)		(2,12)		(1,2)		(2,0)
alpha2	=	a1-x	a1-w-call	b1-x	b1-w-exec	b2	a1-w-exec	a2
sf(alpha2)=		(1,12)	(1,2)	(2,12)	(2,12)	(2,1)	(1,1)	(2,0)
r(alpha2) =		a1		b1		b2		a2
sf(r(alpha1))=		(1,2)		(2,12)		(2,1)		(1,0)
alpha3	=	b1-x	b2	a1-x	a1-w-call			
sf(alpha3)=		(2,12)	(2,1)	(1,1)	(1,0)			
r(alpha3) =		b1	b2	a1				
sf(r(alpha3))=		(2,12)	(2,1)	(1,0)				

Da unser Modell nicht das Ende eines Prozesses vorsieht, wurde hier angenommen, die Aktionen a2, b2 blockierten den aufrufenden Prozeß. Die entstehenden Schaltfolgen würden sonst nicht zu ZFn gehören.

6.2.3. F-System simuliert V-System

Satz:

Sei $K = (M, wk)$ eine Scheduleklasse mit der Relation $wk = wk1$ wie in Abschnitt 6.2.1. definiert.

Sei P ein remanentes V-System, $Q = TFS(P)$. Q simuliert P bezüglich der Klasse K durch die naheliegende Realisierung r :

$$r(a-w) = \lambda, \quad r(a-x) = a.$$

Bemerkung:

Im Gegensatz zum vorherigen Satz ist hier "remanent" eine schwerwiegende Einschränkung: Typische V-Systeme sind ja nicht remanent (siehe 6.2.1.). Dagegen kann hier die Voraussetzung "bipartit" fehlen, weil sie für V-Systeme automatisch gilt.

Die Voraussetzung über die Scheduleklasse ist hier wesentlich schwächer als im vorigen Abschnitt, insbesondere darf M die volle Menge ZFn der zulässigen Schaltfolgen sein.

Korollar:

Sei P ein remanentes V-System, dann simuliert $TFS(P)$ P bezüglich der Klasse ZFn .

Begründung: es ist $ZFn = (M, wk1)$.

Beweis des Satzes:

a) r ist vollständig bezüglich K .

Sei β ein aktiver Ablauf in P ; dann ist α zu finden, so daß

- (1) $r(\text{netto}(\alpha)) = \text{netto}(\beta)$ und
- (2) $\text{sf}(\alpha) \text{ } wk \text{ } \text{sf}(\beta)$.

Hier ist nun Q ein F-System, also $\text{netto}(\alpha) = \alpha$, dagegen unterscheidet sich $\text{netto}(\beta)$ von β durch Fehlen der call-Aktionen.

Die Konstruktion von alpha ist leicht: ersetze im Ablauf beta jede Aktion

a-call durch a-w
 und a-exec durch a-x, falls a-w schon vorher ausgeführt wurde,
 sonst durch die Folge a-w a-x.

(Beachte: die call-Aktion a-call tritt nur dann in beta auf, wenn der betreffende Prozeß blockiert wird - dagegen muß im F-System die Aktion a-w in jedem Fall vor a-x durchlaufen werden - daher die Fallunterscheidung).

Damit ist die Bedingung (1) erfüllt.

alpha ist auch aktiv: wo nämlich a-exec ausführbar ist, da gilt die Vorbedingung von a, d.i. die Fortsetzungsbedingung von a-w.

Um (2) zu zeigen, benutzen wir wieder Induktion:

Induktionshypothese:

Sei $\beta = \beta_1 f$, $\alpha = \alpha_1 g$, und
 $sf(\alpha_1) \text{ wk } sf(\beta_1)$, und zusätzlich
 $NB(\alpha_1) = NB(\beta_1)$.

Dabei ist f eine einzelne Aktion und g die dazu konstruierte Aktionsfolge. Sei $bz(f) = i$, und $NB(\beta_1) = NB_1$.

Gemäß der Konstruktion haben wir drei Fälle zu unterscheiden:

Fall 1. $f = a\text{-call}$, $g = a\text{-w}$.

Dann $sf(\alpha) = sf(\alpha_1) (i, NB_2)$,

$sf(\beta) = sf(\beta_1) (i, NB_2)$,

mit $NB_2 = NB_1 - \{i\}$.

Somit folgt (2) aus der Monoid-Verträglichkeit der Relation wk.

Fall 2. $f = a\text{-exec}$, $g = a\text{-x}$.

Hier gilt

$sf(\alpha) = sf(\alpha_1) (i, NB_2)$ und

$sf(\beta) = sf(\beta_1) (i, NB_2')$.

Falls a-exec Prozesse deblockiert, tut dies auch a-x; falls a-exec jedoch einen fremden Prozeß blockierte (was in V-Systemen möglich ist, siehe 6.1.), dann könnte a-x nicht die gleiche Wirkung haben.

Letzteres ist jedoch durch die Voraussetzung "remanent" ausgeschlossen. Somit gilt $NB2' = NB2$, und (2) folgt mit dem gleichen Argument wie in Fall 1.

Fall 3. $f = a\text{-exec}$, $g = a\text{-w } a\text{-x}$.

Hier ist

$$sf(\alpha) = sf(\alpha_1) (i, NB_1) (i, NB_2),$$

weil $a\text{-w}$ keine Blockade ändert,

und $sf(\beta) = sf(\beta_1) (i, NB_2')$, mit $NB_2' = NB_2$ aus dem gleichen Grund wie in Fall 2. Dann ist aber $NB_1 \subset NB_2$, und nach den Axiomen für wk :

$$\begin{aligned} sf(\alpha) \text{ wk } sf(\alpha_1) (i, NB_2) (i, NB_2) \\ \text{wk } sf(\alpha_1) (i, NB_2) \\ = sf(\beta). \end{aligned}$$

Damit ist die Vollständigkeit gezeigt.

b) r ist sicher bezüglich K .

Sei α ein aktiver Ablauf in Q . Da P ein V-System ist, kommt hier die netto-Funktion zum tragen, und statt direkt $r(\alpha)$ zu betrachten, müssen wir einen aktiven Ablauf β in P so konstruieren, daß

$$(3) \quad r(\text{netto}(\alpha)) = \text{netto}(\beta) \quad \text{und}$$

$$(4) \quad sf(\beta) \text{ wk } sf(\alpha).$$

Da durch (3) die Folge $\text{netto}(\beta)$, also die Teilfolge der exec -Aktionen schon vorgegeben ist, muß die Konstruktion nur noch die call -Aktionen an geeigneten Stellen einfügen. Und zwar muß dies gerade an den Stellen geschehen, wo α eine Aktion $a\text{-w}$ enthält, jedoch nur, falls sich diese Aktion blockiert. Genauer, wir machen wieder die Induktionsannahme:

Es ist $\alpha = \alpha_1 g$, und β_1 ist schon konstruiert, so daß

$$r(\text{netto}(\alpha_1)) = \text{netto}(\beta_1) \quad \text{und}$$

$$sf(\beta_1) \text{ wk } sf(\alpha_1), \text{ und zusätzlich}$$

$$NB(\alpha_1) = NB(\beta_1).$$

Dabei ist g eine Aktion mit $bz(g) = i$.

Nun wird beta konstruiert und gleichzeitig (4) bewiesen.

Fall 1. $g = a-w$ und $NB(\alpha) = NB1$.

D.h. die Aktion a-w blockiert sich nicht. Setze dann

$\beta := \beta_1$,

dann ist $sf(\beta) = sf(\beta_1)$

$wk\ sf(\alpha_1)$

$wk\ sf(\alpha_1)\ (i, NB1)\ \text{(Adsorption)}$

$= sf(\alpha)$

Fall 2. $g = a-w$ und $NB(\alpha) = NB2 = NB1 - \{i\}$,

d.h. die Aktion a-w blockiert sich. Setze dann

$\beta := \beta_1\ a\text{-call}$,

dann ist β aktiv und

$sf(\beta) = sf(\beta_1)\ (i, NB2)$,

weil auch die call-Aktion den Prozeß i blockiert.

Weiter $sf(\alpha) = sf(\alpha_1)\ (i, NB2)$,

somit folgt (2).

Fall 3. $g = a-x$.

Dann setzen wir $\beta := \beta_1\ a\text{-exec}$.

Dies ist aktiv, weil in Q vor der Ausführung von a-x irgendwann die Fortsetzungsbedingung von a-w gegolten hat; diese ist ja die Vorbedingung von a-exec in P; sie gilt wegen der Eigenschaft "remanent" immer noch nach dem Ablauf β_1 .

(Dies ist unabhängig davon, ob in β_1 die Aktion a-call vorkommt oder nicht).

Weiter ist

$sf(\beta) = sf(\beta_1)\ (i, NB2)$

$wk\ sf(\alpha_1)\ (i, NB2)$

$= sf(\alpha)$,

denn die Aktionen a-x und a-exec haben die gleiche Wirkung auf die Blockierungen.

Q.E.D.

6.3. Ein negatives Resultat

Der letzte Abschnitt 6.2.3. gab auf die Frage, wann ein V-System durch ein F-System simuliert werden könne, eine positive Antwort für den Fall eines remanenten Systems.

Wie erwähnt, ist diese Einschränkung aber gravierend: z.B. die üblichen P/V-Operationen erfüllen die Voraussetzung nicht. Daß die Voraussetzung in jenem Satz wirklich notwendig ist, davon überzeugt man sich leicht.

Bisher wurde allerdings nur das Prozeßsystem TFS(P), also die einfachste denkbare Konstruktion eines P simulierenden F-Systems untersucht. Es wäre denkbar, ein gegebenes P auf andere Weise als F-System zu implementieren, und so stellen wir die Frage:

Gibt es zu jedem V-System P ein (wie auch immer geartetes) F-System Q, so daß Q P bezüglich einer geeigneten Schedule-Klasse K simuliert ?

Die Antwort hängt natürlich wesentlich von der Wahl der Klasse K ab.

In diesem Abschnitt zeigen wir an einem Beispiel: für die Schedule-Klasse $K = ZF$ ist die Antwort NEIN. Dann folgt automatisch die negative Antwort auch für größere Klassen.

Wir geben also ein V-System P und zwei zeitfreie Schedules S1 und S2 an, und zeigen, daß kein F-System Q P bezüglich dieser beiden Schedules simuliert; nach dem Satz aus 3.4. kann Q dann P auch nicht bezüglich der Klasse ZF simulieren.

Das Prozeßsystem P ist schon aus Abschnitt 2.2 bekannt; an ihm haben wir den Unterschied zwischen P/V-Operationen und ihrer Siemens-Implementierung studiert:

```
semaphore s init (0);
process P1: loop V(s) ; P(s) endloop;
process P2: loop P(s) ; V(s) endloop;
```

Wir definieren jetzt die nötigen Schedules:

S1 sei der aus 2.2 bekannte nicht-verdrängende Schedule.

S2 wird etwas komplizierter definiert:

1. Ein Prozeß gilt als dringend, wenn er nicht blockiert ist und seit seiner letzten Aktion nacheinander mindestens folgende Folge von Zuständen durchlaufen hat:

nicht blockiert - blockiert - nicht blockiert.

2. Solange es keinen dringenden Prozeß gibt, verdrängt S2 den laufenden Prozeß nicht.

3. Wenn es dringende Prozesse gibt, wird einer von diesen zugeteilt.

Diesen Schedule in unserem Kalkül zu formulieren, wäre zwar umständlich, aber im Prinzip möglich, denn er basiert allein auf der Information, die in den Schaltfolgen codiert ist, nämlich welche Prozesse wann blockiert sind.

Beide Schedules S1 und S2 sind zeitfrei.

Für F-Systeme gilt nun: unter S2 gibt es nie dringende Prozesse, weil ein nicht aktiver Prozeß nicht blockiert werden kann. Daher stimmt das Verhalten eines F-Systems unter S2 mit dem unter S1 überein.

Das gilt nicht für V-Systeme: im Prozeßsystem P ist unter S2 folgender Ablauf aktiv:

$\beta = (P2: P\text{-call}) (P1:V) (P1:P) (P1:V) (P2: P\text{-exec})$

mit $sf(\beta) = (2,1) (1,12) (1,1) (1,12) (2,12)$.

Denn P1 ist hier "dringend" geworden.

Unter dem Schedule S1 ist β nicht aktiv: da sich der Prozeß P1 nicht blockiert, bleibt er aktiv. Mehr noch: kein Ablauf β' in P mit $netto(\beta') = netto(\beta)$ ist unter S1 aktiv.

Wenn nun ein F-System Q das Prozeßsystem P simulierte, dann müßte in Q ein Ablauf α aktiv sein, so daß $r(\alpha) = \text{netto}(\beta)$, und daß α zulässig ist unter S2, aber nicht unter S1; das ist nach dem oben gesagten nicht möglich.

7. Weitere Aspekte der Prozeßkoordinierung

In diesem Abschnitt diskutieren wir verschiedene Aspekte und Probleme paralleler Prozesse, die das formale Modell aus Abschnitt 3. nicht oder nicht vollständig erfaßt. Wir beleuchten dabei praktische Erfahrungen aus der Sicht der Theorie, nämlich mit der Fragestellung:

wie kann man die beobachteten Probleme, Lösungen und Phänomene in einem Modell beschreiben, um dann weitere beweisbare Aussagen zu machen ?

Diese Frage wird jedoch nicht beantwortet; vielmehr sind die folgenden Gedanken als Ausblick und Anstoß für zukünftige Forschungen gedacht.

7.1. Prozeßwechselsperren

Ein in der Praxis häufig angewandtes Mittel der Prozeßkoordinierung sind sogenannte Zustandswechsel-, Prozeßwechsel- oder Interruptsperren. Mit diesen drei Begriffen können verschiedene Dinge gemeint sein; Interrupts im Sinne von peripheren Anforderungen zu sperren, kann bei der E/A-Bearbeitung eines Betriebssystems nötig sein. In Zusammenhang mit parallelen Prozessen meinen wir eine Sperre, die den Prozeßwechsel, also die Verdrängung des laufenden Prozesses verhindert. Stehen dem Anwenderprogramm, wie in Realzeit-Betriebssystemen üblich, Aufrufe wie

"Prozeßwechsel sperren" und "Prozeßwechsel freigeben"

zur Verfügung, bedeutet das eine direkte Beeinflussung des Schedulers auf sehr niedriger Ebene. Der Scheduler arbeitet dann nicht allein mit den Schaltfolgen als Input, sondern mit der zusätzlichen Einschränkung, die die Sperre verlangt.

Eine andere Variante dieser direkten Beeinflussung besteht in Anwenderaufrufen wie "quit" oder "wechsel", mit der Bedeutung: der laufende Prozeß gibt den Prozessor auf; nimmt man an, daß nur bei solchen Aufrufen eine Prozessor-Neuvergabe stattfindet, dann hat der Anwender ebenso direkte Kontrolle über das Scheduling wie mit einer Prozeßwechselsperre.

Häufig wird die Prozessorvergabe auch ähnlich wie in UNIX geregelt: in diesem Betriebssystem werden die Prozesse bei jedem Aufruf an das Betriebssystem (supervisor-Call) neu zugeteilt, sowie zusätzlich bei jedem Sekundenimpuls. Auch dabei hat das Anwenderprogramm eine gewisse Kontrolle über das Scheduling, die über die eigentlichen Synchronisationsoperationen hinausgeht.

Aus Gründen der Verifizierbarkeit, wie auch der Portabilität von Programmen, wäre es wünschenswert, diese Mechanismen allgemeingültig zu beschreiben - oder darauf zu verzichten.

Erst mit einem formalen Modell wäre eine präzise Antwort auf die wichtigen Fragen möglich:

Kann ein Prozeßsystem, das auf einem bestimmten Rechner unter Verwendung der Prozeßwechselsperre und mit Kenntnis der Zuteilstrategie des Betriebssystems entwickelt wurde, auf einen Rechner mit anderer Zuteilstrategie übertragen werden ?
Kann es ohne Prozeßwechselsperre so formuliert werden, daß im Sinne von vollständiger und sicherer Simulation das gleiche Verhalten resultiert ?

Vor dem Hintergrund unseres Konzepts der Scheduleklasse stellt sich die Situation wie folgt dar: Eine gegebene Scheduleklasse legt fest, welche Minimalvoraussetzungen an das Betriebssystem man annehmen darf; sie sagt, in welchen Situationen Prozeßwechsel möglich ist, und welche Information die Zuteilung beeinflusst. Während in unserem Konzept diese Beschreibung lediglich mit Schaltfolgen arbeitet, kann jetzt noch weitere Information hinzukommen, nämlich die genannten Anwenderaufrufe.

Je nachdem, ob Prozeßwechsel jederzeit zulässig sind, oder nur bei expliziten Operationen "wechsel", oder nur bei nicht gesetzter Sperre, u.s.w., ergeben sich wieder unterschiedliche Scheduleklassen - nicht in dem strengen in 3.4. definierten, sondern in einem erweiterten Sinn.

Grundsätzlich wird die Simulation, d.h. die Implementierung eines gegebenen Prozeßsystems umso leichter, je mehr die Scheduleklasse eingeschränkt ist - in diesem Fall: je stärker der Anwender die Prozeßwechsel einschränken kann. So kann man durch explizites Sperren

des Prozeßwechsels im simulierenden System Q verhindern, daß ein Prozeß an einer Stelle unterbrochen wird, wo sein Zustand keinem erlaubten Zustand im simulierten System P entspricht. Der Sachverhalt ist weniger kompliziert, als seine abstrakte Beschreibung befürchten läßt; daher sollen zwei einfache Beispiele das gemeinte verdeutlichen:

Beispiel 1.

Die Idee dieses Beispiel ist schon aus Abschnitt 2.4 bekannt:

Die Folge

(1) $V(s_1); V(s_2)$

von Semaphore-Operationen hat i.a. eine andere Wirkung auf evtl. blockierte fremde Prozesse als die V-multiple-Operation $V(s_1, s_2)$; falls nämlich drei fremde Prozesse an den Operationen $P(s_1)$, $P(s_2)$ und $P(s_1, s_2)$ blockiert sind, kann der letzte Prozeß nur dann deblockiert werden, wenn beide Semaphore gleichzeitig freigegeben werden.

Wird nun die Folge (1) mit den Aufrufen "Prozeßwechsel sperren/freigeben" (PWS/PWF) geklammert, dann kann nach " $V(s_1)$ " noch keine Neuzuteilung stattfinden; nach Ende der Folge ist dann jeder der drei wartenden Prozesse deblockiert, und die Auswahl bleibt dem Scheduler überlassen. Der Effekt von

PWS; $V(s_1)$; $V(s_2)$; PWF;

ist somit derselbe wie von der Operation $V(s_1, s_2)$.

Beispiel 2.

Betrachte die Prozeßsysteme P und Q aus Abschnitt 3.5.

P: process P1: a_1 ; a_2 ;
process P2: b_1 ; b_2 ;
process P3: c_2 ;

Q: process Q1: a_1' ; a_2' ;
process Q2: b_1' ; b_2' ;
process Q3: c_1' ; c_2' ;

mit der Realisierung $r(c_1') = \text{lambda}$, $r(a_1') = a_1$ etc.

Sie haben die Eigenschaft, daß Q P nicht bezüglich des Schedule RR (round robin') simuliert, obwohl sich Q von P nur durch eine unbeob-

achtbare Aktion unterscheidet.

Diese Irregularität wurde durch Einführung der Scheduleklasse SNZ gelöst. Eine andere Lösung besteht in der Verwendung der Prozeßwechselsperre, um die Unterbrechung an der unerwünschten Stelle, nämlich nach der unbeobachtbaren Aktion $c1'$, zu verhindern:

```
Q': process Q1: a1'; a2';
      process Q2: b1'; b2';
      process Q3: PWS; c1'; c2'; PWF;
```

Wir dürfen jetzt mit gutem Grund vermuten: Q' simuliert P - wenn auch diese Behauptung noch keine streng definierte Bedeutung hat, solange nicht auch die Operationen PWS/PWF formalisiert sind.

Die Benutzung der Prozeßwechselsperre erscheint in diesem Beispiel also dann geboten, wenn man nicht voraussetzen darf, daß der Schedule zur Klasse SNZ gehört, wenn man also eine Implementierung mit sehr weitem Gültigkeitsbereich (mit sehr schwachen Annahmen über die Schedule-Strategie) sucht. Umgekehrt kann man auf PWS/PWF verzichten, aber dann gilt " Q simuliert P " nur bezüglich der Klasse SNZ - d.h. Q als Implementierung von P ist vollständig und sicher nur unter stärkeren Voraussetzungen über den Schedule.

7.2. Prozeßkoordinierung in der Praxis

Wie auf vielen Gebieten der Wissenschaft und Technik klafft auch im Bereich der parallelen Programme eine nicht unerhebliche Lücke zwischen Theorie und Praxis. Die Theorie kennt Petri-Netze, conditional critical regions und andere Mechanismen von hohem Niveau - dem Praktiker stehen in der Regel allenfalls Semaphore zur Verfügung. Zwar bilden moderne Programmiersprachen zum Teil Ausnahmen, wie etwa die Sprache ADA mit ihrem ausgefeilten Rendezvous-Konzept, oder auch - in geringerem Grad - die Sprache PEARL mit dem in Abschnitt 4.4. erwähnten bolt-Mechanismus. Gerade in PEARL gibt es aber auch Mechanismen von erheblich niedrigerem Niveau (siehe unten).

Bedenkt man den Anteil der Programme, die auch heute noch in FORTRAN oder Assembler geschrieben werden, dann muß man erkennen, daß eine in die Sprache integrierte Prozeßkoordinierung noch für längere Zeit die (rühmliche!) Ausnahme bleiben wird. Es erscheint deshalb lohnend, eine Bestandsaufnahme der in der Praxis gängigen Methoden zu versuchen.

Hinsichtlich der Implementierung ist zunächst festzustellen: das Fehlen eines Koordinierungskonzepts auf Sprachebene ist keine grundsätzliche Behinderung; das Betriebssystem stellt Aufrufe zur Verfügung, die prinzipiell in jede Sprache in Form von Unterprogrammen oder Makros eingebaut werden können.

Gemeinsame Variable

Die einfachste Art der Programmkommunikation ist das Lesen und Schreiben gemeinsamer Speicherworte, wie es in FORTRAN durch die Common-Bereiche unterstützt wird. Unter Ausnützung der Unteilbarkeit der Zugriffsoperationen läßt sich damit eine gewisse Koordinierung erreichen; z.B. ist das Problem des gegenseitigen Ausschlusses lösbar (DIJK65) - allerdings nur mit aktivem Warten, solange keine Warte-Operation zur Verfügung steht. Dennoch spielen gemeinsame Variable eine gewisse Rolle, insbesondere bei Verwendung der meist vorhandenen Teste-und-setze-Operation.

Dateibelegung

Bei Dateien, die von verschiedenen Prozessen gemeinsam benutzt werden, stellt sich das Problem des exklusiven Zugriffs, für das die Theorie eine Reihe von Lösungen bereit hat; die (theoretisch) einfachste Variante wäre die Klammerung von zusammengehörigen Dateizugriffen eines jeden Programms zu einem kritischen Abschnitt (Transaktion) mittels P/V-Operationen. Eine raffiniertere Lösung wurde unter 2.3. und 4.4. als Leser-Schreiber-Problem diskutiert. Diese Lösungen haben gemeinsam, daß jeder Prozeß am Beginn des kritischen Abschnitts solange blockiert wird, bis ihm der Zugriff erlaubt ist.

In Realzeitsystemen gibt es nun häufig den Fall, daß man die Blockade eines kritischen Prozesses unter keinen Umständen länger als eine gewisse kurze Zeit in Kauf nehmen will. Deshalb wird oft der Weg gewählt, daß ein momentan nicht ausführbarer Aufruf - etwa der Versuch, eine belegte Datei zu belegen - nicht zu einer Blockade, sondern zu einer Fehleranzeige an den aufrufenden Prozeß führt; dieser kann dann selbst über die nötige Reaktion entscheiden. Die Reaktion besteht in vielen Fällen aus einem "retry", d.h. einer Wiederholung des Aufrufs, i.a. nach einer entsprechenden Wartezeit.

Dieses Vorgehen ist intuitiv von niedrigerem Niveau als Semaphore, allerdings kann es nicht ohne weiteres durch Semaphor-Operationen ersetzt werden, weil bei diesen eben die Zeit der Blockierung prinzipiell unbeschränkt ist.

Anhalten und fortsetzen

Eine ebenfalls sehr verbreitete Art der Koordinierung sind Operationen wie

- suspend - der aufrufende Prozeß blockiert sich selbst,
- resume - einen fremden Prozeß fortsetzen,
- activate - einen fremden Prozeß starten,
- terminate - Prozeß beenden.

Diese sind auch in den Sprachen PEARL und Simula (hier allerdings auf Koroutinen bezogen) implementiert.

Dabei handelt es sich also um direkte Einflußnahme auf die Prozeßzustände, im Gegensatz zur indirekten Einflußnahme, wie sie aus höheren Mechanismen resultiert.

Noch eine Stufe weiter gehen die Möglichkeiten, die ein Betriebssystem wie ORG 300 PV dem Anwender bietet; außer der Selbstblockierung eines Prozesses (suspend, hier WARTFO genannt) gibt es die Aufrufe

- anhalten (ANHALT) - Blockieren eines fremden Prozesses, und
- Ende fremd (ENDE) - Beenden eines fremden Prozesses.

Allerdings behandelt das Betriebssystem angehaltene Prozesse anders als sonstige blockierte, und erschwert so eine homogene Beschreibung der Mechanismen.

Mehrfachwarten

Wie schon in Abschnitt 4.2.3 angedeutet, kommen Warte-Funktionen oft in dem Sinn vor, daß alternativ auf eines von mehreren Ereignissen gewartet wird; das Programm soll also fortgesetzt werden, sobald eines dieser Ereignisse eintritt, und zusätzlich muß es zwischen den Alternativen unterscheiden können, um verschiedene Bearbeitungswege anzusteuern.

"Anstoß" von Prozessen

Eine Forderung, die in der Praxis von Realzeitsystemen typisch ist, ist folgende:

- wenn ein Ereignis x eintritt, soll der Prozeß P "sofort" ablaufen. Hierfür ist die Sprechweise üblich "der Prozeß P wird angestoßen" von einem anderen Prozeß. Dies ist mehr als bloße Deblockierung; den Unterschied kann man so charakterisieren:
- | | |
|--------------|------------------------------------|
| deblockieren | - der Prozeß <u>kann</u> ablaufen, |
| anstoßen | - der Prozeß <u>muß</u> ablaufen. |

Eine Präzisierung von "sofort" kann entweder bedeuten "mit höchster Priorität" oder "innerhalb von ... Millisekunden".

Ein höherer Synchronisationsmechanismus - sei er vom Typ "F-System" oder "V-System" - bringt einen Prozeß immer nur in den Zustand "nicht blockiert" bzw. "bereit", und überläßt die tatsächliche Aktivierung dem Scheduler. Bei der Formulierung eines Prozeßsystems mit den Begriffen, wie sie in dieser und anderen theoretischen Arbeiten präzisiert sind, kann man zunächst nicht mehr als einen Prozeß "deblockieren". Damit dieser wirklich aktiv wird, muß man zusätzliche Annahmen über den Scheduler machen - etwa die übliche "Fairness", die aber, wie ich in den Abschnitten 2.4. und 4.3. zu zeigen versucht habe, nicht immer ausreicht.

Das in dieser Arbeit entwickelte Konzept der Betrachtung von Scheduleklassen stellt einen ersten Schritt in dieser Richtung dar. Jedoch lassen sich auch damit die üblichen zeitkritischen Forderungen wie "ein deblockierter Prozeß muß innerhalb von 1500 µsec aktiv werden" nicht formalisieren.

Zeitfunktionen

Während die anderen bisher genannten Aspekte mit den theoretischen Modellen mehr oder weniger gut beschrieben werden können, versagt die Theorie weitgehend, sobald es um die Formulierung von Zeitspannen, statt von bloßen Reihenfolgen geht, wie unter den Stichworten "Dateibelegung" und "Anstoßen" bereits deutlich wurde.

In der Praxis treten noch viele weitere Beispiele von Zeitfunktionen auf:

- das Betriebssystem stellt dem Anwender häufig eine Funktion

"delay(time)"

zur Verfügung; PEARL kennt die Anweisung

"resume after x sec".

- warten auf ein Ereignis, etwa auch das erwähnte Mehrfachwarten, wird sehr häufig zeitlich begrenzt, in dem Sinn: wenn nach (z.B.) 2 Sekunden keines der erwarteten Ereignisse eintrifft, wird das Programm dennoch fortgesetzt, z.B. in einem Fehlerzweig. Dieses Konzept ist in ADA - im select-Konstrukt - erstmals auf hohem Abstraktionsniveau formuliert.

- Wie schon in 2.2. erwähnt, gibt es im Betriebssystem ORG 300 PV "zeitverzögerte Koordinierungsaufrufe"; diese werden zum Beispiel eingesetzt zur Realisierung von zeitlich begrenztem Mehrfachwarten, es gibt aber auch andere Anwendungen. Die Klasse von Prozeßsystemen, die mit einem solchen Mechanismus beschreibbar ist, und vor allem die Frage, wie solche Systeme formal zu spezifizieren sind (Definition der Semantik), wurde meines Wissens bisher nicht untersucht.

Ein formales Modell, das auch diese Zeitfunktionen erfaßt, wäre aber die Voraussetzung dafür, sich auf einer implementierungsfernen Ebene überhaupt zu verständigen, die Korrektheit von Implementierungen zu beweisen, und somit zuverlässige Programme zu produzieren.

Zwar gibt es schon Ansätze eines Kalküls der zeitlichen Logik von Programmen, etwa von Pnueli (PNUE81), aber auch dort beschränkt man sich auf die zeitliche Reihenfolge von Ereignissen und auf Operatoren der Art

"x später als y",

"irgendwann nach Zeitpunkt x gilt Aussage Y",

"immer nach Zeitpunkt x gilt Aussage Y".

Dabei verwendet man eine fiktive diskrete Zeit und muß absehen von der Zeitdauer der einzelnen elementaren Schritte.

Der Grund dafür ist allerdings einsichtig:

gerade die Zeitdauer von einzelnen Operationen ist stark von Implementierungen abhängig, und ist daher eine der ersten Eigenschaften, von der die Theorie abstrahieren muß - eine vernünftige Theorie, die voraussetzt, daß eine Multiplikation genau 0,56 µsec dauert, ist nicht denkbar.

Hier scheint ein echtes Dilemma zu bestehen zwischen notwendiger Abstraktion einerseits und dem Wunsch nach einem Modell für zeitliches Verhalten andererseits.

7.3. Modellierung echter Parallelität

Die Grundlage unseres Modells wie auch der meisten anderen Untersuchungen auf diesem Gebiet bildet die Serialisierung paralleler Abläufe, also die Annahme, daß die einzelnen elementaren Aktionen zeitlich nacheinander auftreten. Wie schon in 2.5. erwähnt, trifft das auf Monoprozessoren recht gut zu, wird aber problematisch im Fall von Multiprozessor-Systemen. Mehrere Prozessoren können ja mehrere Aktionen gleichzeitig ausführen.

Es gibt heute schon diverse Rechnerarchitekturen, die über Einprozessor-Systeme hinausgehen: Arrays von gleichartigen Verarbeitungselementen (Zellen), Pipelines aus verschiedenen Prozessoren, homogene und inhomogene Multiprozessorsysteme (nach GILLOM). Nicht zuletzt durch den bekannten Preisverfall der Hardware ist es abzusehen, daß solche Anordnungen noch an Bedeutung gewinnen werden. Es stellt sich somit die Frage, wieweit die bisherigen Erkenntnisse über Koordinierung paralleler Prozesse auch für den Mehrprozessorfall gelten.

Solange man annimmt, die elementaren Aktionen dauerten keine Zeitspanne, sondern fänden zu einem Zeitpunkt statt, kommt man auch im Mehrprozessorfall noch mit dem Ansatz der Serialisierung aus, wie er dieser und den Arbeiten von Lipton zugrundeliegt. Will man dagegen ausdrücken, daß während der Ausführung einer Aktion auf einem Prozessor eine andere Aktion auf einem anderen Prozessor begonnen wird, dann braucht man ein völlig anderes Modell. Ein solches wurde von Mackert (MACK83) entwickelt; es beruht auf der Betrachtung von drei Zeitpunkten jeder Aktivität:

request, start, end.

Die Zeit zwischen "start" und "end" gibt dann die Dauer der Aktivität wieder, die in unserem ("exec") wie auch in Lipton's Modell zu einem Punkt zusammenschrumpft. Allerdings haben diese zitierten Arbeiten unterschiedliche Zielsetzungen: während bei Lipton wie auch in der vorliegenden Arbeit der Vergleich von Synchronisationsmechanismen auf einer strengen formalen Basis im Mittelpunkt des Interesses steht, zielt Mackert auf die Spezifikation und korrekte Implementierung paralleler Abläufe und die hierfür benötigte Definition der Semantik.

Obwohl die Modellierung von Mehrprozessorsystemen wünschenswert wäre, führt ein Vergleichskonzept (Simulationsbegriff), das auf dem Konzept von request-, start- und end-Zeitpunkt beruht, zu großen Schwierigkeiten:

Zum einen entstehen noch kompliziertere Abläufe, vielfältigere Möglichkeiten als in unserem Modell - das allein wäre natürlich kein Grund, diesen Ansatz nicht zu versuchen.

Ein ernstes Problem stellt dagegen die Abhängigkeit von der Prozessoranzahl dar:

Einerseits soll gewiß ein Synchronisationsproblem P unabhängig von der Zahl der Prozessoren spezifiziert werden, d.h. es wird ein Prozeßsystem angegeben, das eine bestimmte Menge von Folgen von request-, start- und end-Aktionen als aktive Abläufe festlegt. Das Verhalten eines solchen Prozeßsystems kommt zustande aus den Komponenten:

Prozesse + Synchronisation + Scheduler.

Die Semantik eines Synchronisationsmechanismus muß dabei in gewissem Maß unabhängig vom Scheduler und von der Anzahl der Prozessoren definiert sein, etwa im Sinne unseres Simulationsbegriffes: ein Prozeßsystem gilt als Realisierung eines anderen, wenn es unter jedem Schedule (und für jede Prozessoranzahl !?) das gleiche Verhalten zeigt.

Andererseits muß jede Implementierung Q mit einer begrenzten Anzahl m von Prozessoren auskommen - unter den Abläufen sind dann aber nur solche aktiv, die der Bedingung genügen: "zu jedem Zeitpunkt sind höchstens m Prozesse aktiv", oder formal ausgedrückt:

(1) $\#start - \#end \leq m$,

wobei #start und #end die Anzahlen sämtlicher gestarteteter bzw. beendeter Aktionen bedeuten.

Damit wird es äußerst fraglich, ob eine vollständige Realisierung überhaupt noch möglich ist; falls das zu implementierende System P nicht schon der Bedingung (1) genügt, kann man das kaum behaupten.

Man hat hier wieder die alte Frage nach der Trennungslinie zwischen Synchronisationsmechanismus und Scheduler - es gibt in diesem Zusammenhang unterschiedliche Antworten:

1. Die Synchronisationsoperationen definieren die Prozeßzustände blockiert/ nicht blockiert. Sie sind unabhängig von der Prozessoranzahl. Der Scheduler wählt unter den nicht blockierten Prozessen welche aus, die aktiv werden. Er ist dabei durch die Prozessoranzahl eingeschränkt, d.h. er realisiert ein Multiplexing der Prozessoren. Diese Antwort führt zum erwähnten Verlust der Vollständigkeit, zumindest wenn man das Verhalten eines Prozeßsystem auffaßt als resultierend aus den Komponenten:

Prozesse + Synchronisation + Schedule.

2. Die Synchronisationsoperationen definieren die Prozeßzustände blockiert/ bereit/ gestartet. Sie sind durch die Prozessoranzahl eingeschränkt. Der Scheduler spielt dann keine Rolle. Diese Antwort widerspricht aber der Auffassung, daß Synchronisation auf einer höheren Ebene und unabhängig von zufälligen Gegebenheiten des Betriebssystems definierbar sein müssen.

3. Die Synchronisationsoperationen definieren die Prozeßzustände blockiert/ bereit/ gestartet. Sie sind durch die Prozessoranzahl nicht eingeschränkt, d.h. es können beliebig viele Prozesse "gestartet" werden, diese sind jedoch noch nicht wirklich aktiv. Der Scheduler wählt dann unter den "gestarteten" Prozessen welche aus, die aktiv werden. Er ist dabei durch die Prozessoranzahl eingeschränkt.

Diese Antwort ist nicht nur am kompliziertesten; sie vermeidet nur dann den Nachteil der ersten Antwort, wenn der Simulationsbegriff ebenfalls den Unterschied zwischen "gestartet" und "aktiv" ignoriert - d.h. wenn er nur das Verhalten der

Prozesse + Synchronisation (ohne Scheduler)
berücksichtigt. Es gilt dann auch nicht die Bedingung (1).

Keiner dieser drei Standpunkte wird also der tatsächlichen Situation gerecht. Ich habe aus diesem Dilemma keinen Ausweg anzubieten; es zeigt aber immerhin die Schwierigkeit der Modellierung echter Parallelität, und rechtfertigt damit einen vereinfachenden Ansatz wie die vorliegende Arbeit.

8. Zusammenfassung

Als wesentliche Ergebnisse dieser Arbeit sind zwei Punkte zu nennen: zum einen der Komplex "F-Systeme", der den Ausgangspunkt der Untersuchungen bildete und zum anderen das Konzept der Scheduleklassen mit dem sich daraus ergebenden Lösungsbegriff.

F-Systeme

Ohne die Ergebnisse im einzelnen zu wiederholen, läßt sich festhalten: F- und V-Systeme stehen als gleichwertige Konzepte der Prozeßsynchronisation nebeneinander; Abschnitt 6. ergab, daß keines der beiden das andere in jedem Fall ersetzen kann.

Diese allgemeine Erkenntnis wird auch gestützt von den Beispielen des Abschnitts 4.: die angegebene Lösung des Philosophenproblems als F-System etwa erscheint uns eleganter und besser verständlich als die verschiedenen aus der Literatur bekannten Lösungen als V-System. Dieses sowie die anderen Beispiele zu Belegungsproblemen (Abschnitt 4.3.), aber auch zum Leser-Schreiber-Problem deuten darauf hin, daß F-Systeme dann besser geeignet sind, wenn die Frage der Fairness interessiert, also die Vermeidung des Aushungerns einzelner Prozesse. Satz 2 aus 4.3.2. bestätigt diese Vermutung.

Das hängt wiederum damit zusammen, daß in der Definition eines Synchronisationsmechanismus als F-System schon mehr Strategie-Entscheidung enthalten ist als bei V-Systemen; im Standardbeispiel der Semaphore bedeutet das: während P/V-Operationen offen lassen, welchen Prozeß von mehreren wartenden eine V-Operation deblockiert, legt der ANFO/FREI-Mechanismus dies fest.

Zwar ist ANFO/FREI nicht die einzige mögliche Entsprechung zu P/V-Operationen als F-System, aber wohl die natürlichste und am leichtesten zu definieren. Auch ist ANFO/FREI (unter den Voraussetzungen von 4.3.2.) aushungerungsfrei, und dies steht im Einklang mit obiger Feststellung: eine nicht aushungerungsfreie Implementierung von P/V-Operationen als F-System zu definieren, wäre umständlicher und weniger natürlich. Das Wort "Implementierung" kann man allerdings hier nur mit großem Vorbehalt gebrauchen, denn wir wissen, daß eine Realisierung z.B. der

P/V-Operationen durch ein F-System nicht zugleich sicher und vollständig sein kann (Abschnitt 6.3.) ! Tatsächlich sind die zur Diskussion stehenden Mechanismen (wie ANFO/FREI) alle sicher, aber nicht vollständig - das heißt eben gerade "mehr Strategie-Entscheidung".

Das alles bedeutet auch größere Implementierungsnähe - ja man darf sagen, daß die Untersuchung von F-Systemen, wie in Abschnitt 2. dargelegt, eigentlich überwiegend aus dem Studium von implementierten Koordinierungsmechanismen motiviert wurde. Dem Theoretiker mag das zunächst suspekt erscheinen, und tatsächlich soll nach dem Stand des Software Engineering der Entwurf (Spezifikation) eines Programmsystems sich unbedingt am Problem orientieren, möglichst unbeeinflußt von späteren Gesichtspunkten der Implementierung.

Ist also das Konzept "F-System" ungeeignet, wenn eine homogene Methode zur Softwareentwicklung in allen Phasen gesucht wird ? Vorweg: die vorliegende Arbeit beansprucht nicht, eine solche Methode zu bieten; aber sie will wohl zur theoretischen Fundierung beitragen.

Die Antwort auf die genannten Zweifel muß heißen: Weil eine gegebenes V-System nur unter sehr starken Einschränkungen (insbesondere Remanenz, siehe 6.2.1., 6.2.3. und 6.3.) von einem F-System simuliert, d.h. vollständig und sicher implementiert werden kann, wäre es einfach sinnlos, einen Entwurf als V-System zu formulieren, wenn für die Implementierung nur F-Systeme zur Verfügung stehen. Auswege gibt es mehrere:

- entweder man versucht die Implementierung von V-Systemen; eine solche ist mir bisher nicht bekannt; Abschnitt 2. legt nahe, daß hier prinzipielle Schwierigkeiten auftreten - jedoch ist auch die Unmöglichkeit keineswegs erwiesen.

- oder man verzichtet auf die Vollständigkeit - ein durchaus nicht absurder Vorschlag, der sich auch mit der derzeitigen Praxis deckt ! Es wäre ein Simulationsbegriff zu suchen, der statt Vollständigkeit eine schwächere Eigenschaft fordert.

- Schließlich bleibt als theoretisch sauberste Lösung die gemeinsame Verwendung von F-Systemen für Entwurf und Implementierung.

Zu den Fragen, die bisher offen bleiben, gehört auch die nach "VF-Systemen"; damit meinen wir die Vereinigung der beiden Konzepte, also Prozeßsysteme mit Vor- und Fortsetzungsbedingungen: welches Verhalten läßt sich damit beschreiben ? Gibt es Aufgaben, die ein bestimmtes VF-System besser löst als es die einfacheren Konzepte tun ?

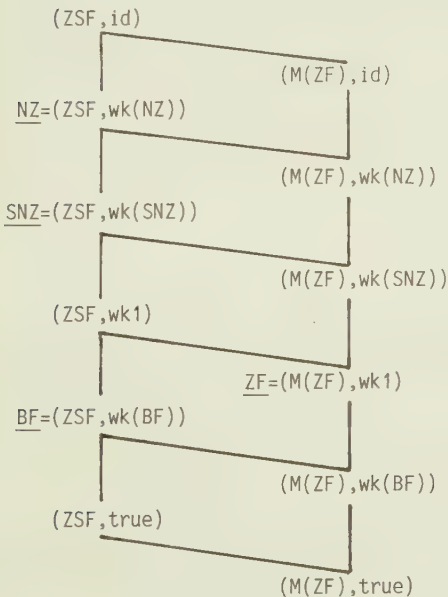
Scheduleklassen

Vor einem Resumé unseres Schedule-Konzepts seien hier die diversen in dieser Arbeit definierten Scheduleklassen zusammengestellt: diese sind in natürlicher Weise partiell geordnet vermöge der Relation "umfaßt":

$S=(M, wk)$ umfaßt $S'=(M', wk')$, falls gilt: $M \supset M'$ und $wk \subset wk'$.

Um die verschiedenen Klassen zu bezeichnen, schreiben wir

$S=(M(S), wk(S))$ für jede definierte Klasse S . So ist z.B. $M(ZF)$ die für die Klasse $ZF=(M, wk)$ definierte Menge M .



Bemerkungen (Abschnitt) :

id = Identität
= feinste Relation

nicht-zählend (3.3.)

schwach nicht-zählend (3.5.)

wk1 = Relation aus (6.2.1.)

zeitfrei

blockadefrei (5.2.)

true = gröbste Relation

In der vorstehenden Übersicht stehen nur die unterstrichenen Bezeichnungen für Scheduleklassen, die wir explizit definiert haben, die anderen aufgeführten ergeben sich aus Kombinationen der diversen Relationen wk mit den beiden Mengen $M=ZSF$ (= alle zulässigen Schaltfolgen) oder $M=M(ZF)$ (= Schaltfolgen ohne Zeitunterbrechung). Gewiß konnte man noch weitere interessante Mengen M finden, ebenso weitere Relationen wk ; andererseits wird auch nicht behauptet, alle hier aufgeführten Scheduleklassen hätten interessante Eigenschaften; immerhin gilt das für BF (Abschnitt 5.) und ZF (Abschnitt 6.).

Aber nicht bestimmte Scheduleklassen zu untersuchen, ist unser Hauptanliegen, sondern die Nützlichkeit des Konzepts als solchen zu zeigen. Daher sei nochmals zusammengefaßt, was über den anschaulichen Sinn dieses Begriffs schon an verschiedenen Stellen angedeutet wurde (Abschnitte 6., 7.1.).

Ausgehend von der Überlegung, daß das Gesamtverhalten eines Programmsystems außer von den einzelnen Programmen und ihrer Koordinierung auch von der Scheduling-Strategie des Betriebssystems abhängt, stellt man fest: beim Entwurf und der Entwicklung eines portablen Programms muß man genau die Annahmen festlegen, die man über das Betriebssystem machen will; sind diese sehr stark, also auf ein bestimmtes Betriebssystem zugeschnitten, kann von Portabilität keine Rede sein - dies entspräche einer sehr engen (in obiger Hierarchie niedrigen) Scheduleklasse, oder einem einzigen Schedule.

So sind etwa die Scheduleklassen mit

$wk=true$

wie folgt zu verstehen: Der Scheduler wird vom Verhalten der Prozesse, also von eventuellen Blockierungen, gar nicht beeinflußt. Der zugeordnete Simulationsbegriff bedeutet bloße Entsprechung der Aktionenfolgen selbst, ohne Berücksichtigung von Blockaden; d.i. in Lipton's Begriffen "vollständige und sichere" Simulation, ohne "blockadefrei".

Bezüglich dieses groben Simulationsbegriffs verschwinden bekanntlich die Unterschiede zwischen verschiedenen Synchronisationsmechanismen - die P/V-Operationen können alle anderen ersetzen. Das gleiche dürfte auch für die Unterschiede zwischen F- und V-Systemen gelten.

Sehr schwache Annahmen über das Betriebssystem führen dagegen zu einem weiten Bereich von zulässigen Zielrechnern, zu einem großen Geltungsbereich des Programms. Allerdings heißt das eben, das Prozeßsystem muß auch unter einer sehr großen Zahl von möglichen Schedules das gewünschte Verhalten zeigen - diese Voraussetzung ist so schwer zu erfüllen, wie der Simulationsbegriff bezüglich einer weiten (in der Hierarchie hohen) Klasse streng ist. So zeigten diverse Beispiele (3.3., 3.5.), daß eine zu feine Relation w_k zu einem fast unerfüllbaren Simulationsbegriff führt, genauer: ein Prozeßsystem P kann dann nur noch von einem isomorphen Prozeßsystem Q simuliert werden.

Desgleichen zeigt Abschnitt 6.2.: ohne Einschränkung der Menge M (hier auf zeitfreie Schedules) kommt man in gewissen Fällen nicht aus - genauer, gewisse Implementierungen funktionieren nur für Schedules ohne Zeitscheibenunterbrechung.

Bezüglich der Hierarchie der Synchronisationsmechanismen ist festzuhalten: je höher die dem Vergleichsbegriff zugrundeliegende Scheduleklasse, desto feiner wird die Hierarchie.

Bei der Auswahl eines Synchronisationsmechanismus ist also auch zu bedenken, welche Annahmen man über die Scheduling-Strategie machen darf oder will.

Literaturverzeichnis

- (ANDL79) S. Andler
 "Predicate Path Expressions: A High-Level Synchronization Mechanism"
 Ph. D. thesis, Carnegie-Mellon-University, Pittsburgh, 1979
- (ANDR81) Andrews
 "Parallel Programs: Proofs, Principles, and Practice"
 CACM 24(3), 1981
- (ASHC75) Ashcroft
 "Proving Assertions About Parallel Programs"
 Journal of Computer and System Science, 10, 1975, S. 110-135
- (BATH82) P. Bathelt
 "Vergleich von Synchronisationsmechanismen"
 IMMD-Bericht 15, 3, 1982
- (BET070) C. Betourné et al.
 "Process Management and Resource Sharing in the Multiaccess System ESOPÉ"
 CACM 13(12), 1970, S.727ff.
- (COUR71) P.J.Courtois, F. Heymans, D.L.Parnas
 "Concurrent Control with Readers and Writers"
 CACM 15, 1971, S. 667-668
- (COUR72) P.J.Courtois, F. Heymans, D.L.Parnas
 "Comments on 'A Comparison of Two Synchronizing Concepts' by P.B. Hansen"
 Acta Informatica, 1, 1972, S. 375- 376
- (DIJK65) E.W. Dijkstra
 "Solution of a Problem in Concurrent Programming Control"
 CACM 8(9), 1965, S. 569f.
- (DIJK68) E.W. Dijkstra
 "Cooperating Sequential Processes"
 in "Programming Languages", ed. F. Genuys, S. 43-112,
 Academic Press, New York, 1968
- (DIJK72) E.W. Dijkstra
 "Hierarchical Orderings of Sequential Processes"
 in "Operating Systems Techniques", ed. Hoare, Perrott, Academic Press London - New York 1972
 (auch in Acta Informatica, 1, 1971, S. 115-138)

- (DIJK75) E.W. Dijkstra
 "Guarded Commands, Nondeterminacy and Formal Derivation of Programs"
 CACM 18(8), 1975, S. 453-457
- (DOLE77) D. Dolev
 "Local Characterization of Models of Synchronisation Primitives"
 Proc. of Waterloo Conference on Theoretical Computer Science, University of Waterloo, Waterloo, Ontario, Canada, August 1977, S. 53-60
- (DOLE79) D. Dolev
 "Commutation Properties and Generating Sets Characterize Slices of various Synchronization Primitives"
 Theor. Computer Science, 8, 1979, S. 379-391
- (GIL081) W.K. Giloi
 "Rechnerarchitektur"
 Springer-Verlag Berlin, Heidelberg, New York, 1981
- (GLIG80) U.D. Gligor, S.H. Shattuck
 "Deadlock Detection in Distributed Systems"
 IEEE Trans. on Software Engineering 6,5, 1980
- (GRIE77) D. Gries
 "An Exercise in Proving Parallel Programs Correct"
 CACM 20(12), 1977
- (HABE72) A.N. Habermann
 "Synchronization of Communicating Processes"
 CACM 19(3), 1972, S. 171-176
- (HANS72) P. Brinch Hansen
 "A Comparison of Two Synchronizing Concepts"
 Acta Informatica, 1, 1972, S. 190-199
- (HANS73) P. Brinch Hansen
 "A Reply to Comments on 'A Comparison of Two Synchronizing Concepts'"
 Acta Informatica, 2, 1973, S. 189-190
- (HANS78) P. Brinch Hansen
 "Distributed Processes: A Concurrent Programming Concept"
 CACM 21, 1978, S. 934-941

- (HEPI81) R. Herschel, F. Pieper
 "PASCAL: Systematische Darstellung von PASCAL und CONCURRENT
 PASCAL für den Anwender"
 Oldenburg Verlag München Wien 1981

- (HOAR74) C.A.R. Hoare
 "Monitors: an Operating System Structuring Concept"
 CACM 17(10), 1974, S. 549-557

- (HOAR78) C.A.R. Hoare
 "Communicating Sequential Processes"
 CACM 21(8), 1978, S. 666-677

- (HOFM74) F. Hofmann
 "Betriebsprogrammierung "
 Hrsg. Bolch, Keramidis; Vorlesungsskript Erlangen 1974

- (HOFM80) F. Hofmann
 "Betriebsprogrammierung I+II"
 Vorlesung Erlangen 1980

- (HOFM82) F. Hofmann
 "Koordinierung asynchroner Prozesse"
 Vorlesung Erlangen 1982

- (KAMR79) Kappatsch, Mittendorf, Rieder
 "PEARL, Systematische Darstellung für den Anwender"
 Oldenburg Verlag, München, 1979

- (KELL76) Keller
 "Formal Verification of Parallel Programs"
 CACM 19(7), 1976

- (KEMA79) S. Keramidis, L. Mackert
 "Specification and Implementation of Parallel Activities on
 Abstract Objects"
 Proc. 4th Int. Conf. on Software Engineering, 1979,
 S. 203-211

- (KEMA80) S. Keramidis, L. Mackert
 "Ein Kalkül zur Konstruktion deadlockfreier Systeme"
 GI-NTG Fachtagung "Struktur und Betrieb von Rechensystemen",
 Kiel 1980

- (KERA82) S. Keramidis
 "Eine Methode zur Spezifikation und korrekten Implementierung
 von asynchronen Systemen"
 IMMD-Bericht, 15, 4, 1982
- (KERN82) B.W. Kernighan, D. M. Ritchie
 "UNIX Programming - Second Edition"
 Bell Laboratories, New Jersey
 in: UNIX-Manual für Anfänger, Hrsg. C.U. Linster,
 IMMD IV, 1982
- (LAMP80) Lamport
 "The Hoare Logic of Concurrent Programs"
 Acta Informatica 14, 1980, S. 21-37
- (LAVE78) M.S. Laventhal
 "Synthesis of Synchronization Code for Data Abstractions"
 PhD Thesis, MIT, Lab. for Computer Sciences, 1978
- (LAVE79) M.S. Laventhal
 "A Constructive Approach to Reliable Synchronization Code"
 Proc. 4th Int. Conf. on Software Engineering, 1979,
 S. 194-202
- (LIPT74) R.J. Lipton, L. Snyder, Y. Zalcstein
 "A Comparative Study of Models of Parallel Computation"
 15th Annual Symposium on Switching and Automata Theory,
 New Orleans, Oct. 1974, S. 145-155
- (LIPT74') R.J. Lipton
 "Limitations of Synchronization Primitives"
 Research Report #31, Comp. Science Dep., Yale University
 August 1974
- (LIPT75) R.J. Lipton
 "Reduction: A Method of Proving Properties of Parallel
 Processes"
 CACM 18(12), 1975
- (MACK78) L. Mackert
 "Spezifikationsmethoden für Systemsoftware"
 Studienarbeit am IMMD IV, Erlangen, 1978
- (MAKE82) L. Mackert, S. Keramidis
 "A Formal Method for Specifying Hardware Modules"
 Proc. 8th EUROMICRO Symp., 1982, S. 229-239

- (MACK83) L. Mackert
 "Modellierung, Spezifikation und korrekte Realisierung
 von asynchronen Systemen"
 IMMD-Bericht 16, 7, 1983

- (ORG300) "ORG 300 PV Beschreibung"
 Programmservice Siemens Systeme 300
 Best.Nr. P71100-B0340-X-X-35, Band 1 und 2

- (OWIC76) S. Owicki, D. Gries
 "Verifying Properties of Parallel Programs: An Axiomatic
 Approach"
 CACM 19(5), 1976, S. 279-285

- (Piep77) F. Pieper
 "Einführung in die Programmierung paralleler Prozesse"
 R. Oldenburg Verlag GmbH, München 1977

- (PNUE81) Pnueli
 "The Temporal Semantics of Concurrent Programs"
 Theoretical Computer Science, 13, 1981, S. 45-60

- (PYLE82) I.C. Pyle
 "Die Programmiersprache ADA"
 Carl Hanser Verlag München
 (engl. Original bei Prentice-Hall Int. Inc., London 1981)

- (ROHL73) H. Rohlfing
 "SIMULA - Eine Einführung"
 BI - Hochschultaschenbücher, Band 747, 1973

- (WETT78) H. Wettstein
 "Aufbau und Struktur von Betriebssystemen"
 Carl Hanser Verlag München Wien 1978

- (WINK82) K. Winkelmann
 "Ein Beispiel eines Thrashing-Effekts durch ungeschickte
 Prozeßsynchronisation"
 interne Mitteilung, Erlangen 1982

Abkürzungen und Symbole

Name ist Schall und Rauch.
Goethe, Faust.

Beim Schreiben dieser Arbeit habe ich einige Zugeständnisse an die elektronische Textverarbeitung gemacht und den Zeichenvorrat etwas eingeschränkt.

Insbesondere wurden Folgen statt wie üblich mit griechischen Buchstaben, mit deren (in lateinischer Schrift ausgeschriebenen) Namen bezeichnet, also alpha/beta/lambda statt $\alpha / \beta / \lambda$.

Auch auf die Tiefstellung von Indizes habe ich verzichtet, und statt

$A_1, A_2, \dots, A_i, A_n$
geschrieben

$A1, A2, \dots, Ai, An,$
oder wo dies mißverständlich schien:

$A(1), A(2), \dots, A(i), A(n).$

Der Leser möge diese Bequemlichkeit verzeihen und sich hoffentlich nicht durch ungewohnte Bezeichnungen irritieren lassen.

Weiter verwenden wir folgende Bezeichnungen:

Übliches Zeichen	Verwendete Zeichenkette	Bedeutung
$\emptyset$	0	leere Menge
$\neg$	not	Negation
$\wedge$	and, &	logische Konjunktion
$\vee$	or	logische Disjunktion
$\Leftrightarrow$	g.d.w.	logische Äquivalenz, genau dann wenn
$\equiv$	äq	allgemeine Äquivalenz
$\mathcal{P}(M)$	Pow(M)	Potenzmenge
$M \rightarrow N$	Abb(M,N)	Menge der Abbildungen von M nach N
A^*	Star(A)	Menge der Folgen über A
λ	lambda	leere Folge
$\mathbb{Z}$	Z	Menge der ganzen Zahlen

Weitere Bezeichnungen, die in dieser Arbeit eingeführt wurden:

Abkürzung	Einführung in Abschnitt:	Bedeutung
act(.)	3.1.	Menge der aktiven Abläufe
BFn	5.2.	Scheduleklasse "blockadefrei"
bz(.)	1.1., 3.1.	"Befehlszähler" = Prozeßnummer
-call	3.2.	Aufrufzeitpunkt einer Aktion
-exec	3.2.	Ausführungszeitpunkt einer Aktion
NB	3.1.	Menge nicht blockierter Prozesse
NB(.)	3.6.	dito, als Funktion des Ablaufs
netto(.)	5.1.	Funktion: Weglassen von "call"
NZn	3.3.	Scheduleklasse "nicht-zählend"
Perm(n)	3.6.	Permutationen (symmetr. Gruppe)
r(.)	1.1., 3.1.	Realisierungsfunktion
SFn	3.3.	Menge der Schaltfolgen
sf(.)	3.1.	Schaltfolge eines Ablaufs
SNZn	3.5.	Sched.-kl. "schwach nicht-zählend"
wk	3.3.	"weaker"-Relation der Schaltfolgen
ZFn	6.2.1.	Scheduleklasse "zeitfrei"
ZSFn	3.3.	Menge zulässiger Schaltfolgen

Lebenslauf
Klaus Winkelmann

- 5.6.1954 geboren in Bamberg, Vater Dr. Eduard Winkelmann, Arzt,
Mutter Elisabeth-Charlotte Winkelmann
- 1964-73 Kaiser-Heinrich-Gymnasium Bamberg, Humanistisches Gymnasium;
Leistungskurse Griechisch und Mathematik; weitere Fremdspra-
chen Latein, Englisch, Französisch, Russisch, später
Spanisch.
- 1971-73 Mehrere Preise im Bundeswettbewerb Mathematik.
- 1973 Abitur
- 1973-74 Bundeswehr
- 1974-79 Studium der Mathematik in Erlangen, Nebenfach Informatik,
Spezialgebiet Kombinatorik; weitere Schwerpunkte sind Algebra
und Automatentheorie.
- dazwischen
1977-78 Auslandsstipendium des DAAD: am Queen-Mary-College, London
- November 1979: Diplomprüfung mit Auszeichnung,
Diplomarbeit bei Prof. Leeb "Das verallgemeinerte
Wägeproblem".
- Dezember 1979 - Juli 1981: freier Mitarbeiter bei Prodata Software
GmbH, Erlangen.
Problemanalysen, Programmierung, Inbetriebnahme, überwiegend
im Bereich Prozeßrechner, Graphik-Software
- August 1981 - Dezember 1982: wissenschaftlicher Mitarbeiter bei
mbp, Mathematischer Beratungs- und Programmierungsdienst,
Dortmund und Erlangen im Bereich Realzeitdatenverarbeitung,
Automation, Fertigungssteuerung
- seit Sommer 1982: Kontakte zum IMMD Erlangen, Lehrstuhl 4,
Vorbereitung auf die Promotion
- seit April 1983: freier Mitarbeiter bei Siemens, Zweigniederlassung
Nürnberg, Bereich VES - Vertrieb Energietechnik Systeme.

